Investigating Advances in the Acquisition of Systems Based on Open Architecture and Open Source Software

Scacchi, Walt; Alspaugh, Thomas A.; Asuncion, Hazel

Monterey, California. Naval Postgraduate School

UCI-AM-11-164

Investigating Advances in the Acquisition of Systems Based on Open Architecture and Open Source Software

Grant #N00244-10-1-0038

Walt Scacchi, Thomas A. Alspaugh and Hazel Asuncion
Institute for Software Research
University of California, Irvine
Irvine, CA 92697-3455 USA
{wscacchi, alspaugh}@ics.uci.edu

Final Report
August 1 2011

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

UCI-AM-11-164 i
Investigating Advances in the Acquisition of Systems Based on Open Architecture and Open Source Software i
1. Executive Summary 7
1.1. Research Description 7
1.1.1. Problem for Acquisition Research 8
1.1.2. Issues for Acquisition Research 9
1.1.3. Prospects for longer-term Acquisition-related research 10
1.2. Acquisition Research Results in this Report 10
1.3. Research Going Forward 11
1.4. References 13
2. Software Licenses in Context: The Challenge of Heterogeneously-Licensed Systems 15
Abstract 15
2.1. Introduction 16
2.2. A Motivating Example 21
2.3. Related Research 22
2.4. Intellectual Property (IP) Basics 24
2.5. Software Licensing and FOSS Licenses 25
2.6. A Theory of Software Licenses and their Application 26
2.6.1. Legal Grounding 26
2.6.2. A Basic Meta-Model for Licenses 27
2.6.3. Reasoning Rules in this Meta-Model 28
2.6.4. Extending the Theory Using Empirical Data 29
2.7. License Architecture of Heterogeneously-Licensed Systems 35
2.7.1. Components 36
2.7.2. Connectors 36
2.7.3. Other Information in a License Architecture 37
2.8. Embodying and Applying the Models and Analysis 38
2.9. Discussion 43
2.10. Conclusion 44
Acknowledgments xlvi
2.11. References 49
3. Understanding the Role of Licenses and Evolution in Open Architecture Software Ecosystems 55
Abstract 55
3.1. Introduction 56
3.1.1. A Motivating Example 57
3.1.2. Understanding Open Architecture Software Ecosystems 58
3.2. Related Research 63
3.3. Open Architectures 65
3.4. Software Licenses 71
3.5. Architecture, License, and Ecosystem Evolution 74
3.6. Discussion 77

3.7. Conclusion 79
Acknowledgments....80
3.8. References....81
4. Intellectual Property Rights Requirements for Heterogeneously-Licensed Systems85
Abstract 85
4.1. Introduction 86
4.2. A motivating example 88
4.3. Background 88
4.3.1. Intellectual Property (IP) basics 88
4.3.2. Open Source Software (OSS)....89
4.3.3. Open Architecture (OA)....90
4.4. Related work 93
4.5. Analyzing software licenses 94
4.6. Analyzing license architectures 100
4.7. Automating the analysis 101
4.8. Discussion....103
4.9. Conclusion 105
Acknowledgments....107
4.10. References....109
5. The Future of Research in Free/Open Source Software Development 111
Abstract 111
5.1. Introduction 112
5.2. Sample Research Results on FOSSD 112
5.3. Research Opportunities for FOSSD and SE 114
5.4. Broader Impact Areas for FOSS Research and Development 116
5.4.1. Software Development....116
5.4.2. Science and Industry 118
5.5. Conclusions 119
Acknowledgments....121
5.6. References....123
6. Free/Open Source Software Development 125
Abstract 125
6.1. Introduction 126
6.2. What is free/open source software development? 127
6.2.1. The Early Days of Software Source Code Sharing 128
6.2.2. The Rise of Empirical Studies of FOSSD....129
6.3. Understanding FLOSS development across different communities 130
6.3.1. Networked computer game worlds....131
6.3.2. Internet/Web infrastructure 132
6.3.3. Scientific computing in X-ray astronomy and deep space imaging 133
6.3.4. Academic software systems design 134
6.3.5. Overall characteristics across different FOSSD communities 134
6.4. Informalisms for describing FLOSS requirements and design rationale....136
6.5. FOSSD Research Studies and Approaches....141
6.5.1. Objects of study, success indicators, and critical factors for understanding FOSSD 142
6.5.2. Current FOSSD research approaches 142

6.6. Opportunities for FOSSD and SE research and practice 144
6.7. Limitations and Conclusions on FOSSD 146
Acknowledgments.....151
6.8. References.....153
7. Final Report Conclusions 157
7.1. Introduction 157
7.2. Emerging research opportunities that follow from our research results.....159
Acknowledgments.....clxii
7.3. References.....165

THIS PAGE INTENTIONALLY LEFT BLANK

1. Executive Summary

In 2007–08, we began an investigation of problems, issues, and opportunities that arise during the acquisition of software systems that rely on open architectures and open source software. The current effort funded for 2009–10 (funding received in 2010) seeks to continue and build on the results in this area while refining its focus to center on the essential constraints and tradeoffs we have identified for software-intensive systems with open architecture (OA) and continuously evolving open source software (OSS) elements. The U.S. Air Force, Army, and Navy have all committed to an open technology development strategy that encourages the acquisition of software systems whose requirements include the development or composition of an OA for such systems, and the use of OSS systems, components, or development processes when appropriate. Our goal is to further develop and document foundations for emerging policy and guidance for acquiring software systems that require OA and that incorporate OSS elements.

This report documents and describes the findings and results that we have produced as a result of our research into the area of the acquisition of software systems that rely on OA and OSS. In particular, it includes two research papers that have been refereed, reviewed, presented, and published in national and international research conferences, symposia, and workshops. The first of these papers has been published in the *Journal of the Association for Information Systems* (JAIS), a top-tier international academic research journal, while the second was originally presented at the 2010 Acquisition Research Symposium in May 2010 and is also under consideration for publication in another international research journal. Three other refereed research papers were also produced, two were included at international research conferences, and one as an invited book chapter. All five of these research publications are included in this report.

1.1. Research Description

Open source software (OSS) is an integrated web of people, processes, and organizations, including project teams operating as virtual organizations [Scacchi 2007]. There is a basic need to understand how to identify an optimal mix of OSS within Open Architectures (OA) as products, production processes, practices, community activities, and multi-project (or multi-organization) software ecosystems. However, the relationship among OA, OSS, requirements, and acquisition is poorly understood [cf. Scacchi 2009, Naegle and Petross 2007]. Subsequently, in 2007–08, we began by examining how different OSS licenses can encumber software systems within OA, which

therefore gives rise to new requirements for how best to acquire software-intensive systems with OA and OSS elements [Scacchi and Alspaugh 2008]. Following this starting point, we have further investigated emerging requirements for the acquisition of OA systems that involve systems or components that are subject to different contractual obligations and rights specified in their associated software property licenses [Alspaugh, Asuncion, and Scacchi 2009a,b,c, 2010, Asuncion 2009].

1.1.1. Problem for Acquisition Research

OA seems to imply software system architectures incorporating OSS components and open application program interfaces (APIs), while also conforming to open standards. But not all software system architectures incorporating OSS components, open APIs, and open standards will produce OA [cf. Scacchi and Alspaugh 2008] because OA depends on (a) how/why OSS and open APIs are located within the system architecture, (b) how OSS and open APIs are implemented, embedded, or interconnected, and (c) whether the copyright licenses assigned to different OSS components encumber all/part of a software system's architecture into which they are integrated. Similarly, (d) alternative architectural configurations and APIs for a given system may or may not produce an OA at design-time, build-time, or release/run-time [Alspaugh, Asuncion, Scacchi 2009a,b].

Subsequently, we believe this can lead to situations in which if program acquisition stipulates a software-intensive system with an OA and OSS, and the architectural design of a system constrains system requirements (i.e., what requirements can be satisfied by a given system architecture, or given system requirements what architecture is implied), then the resulting software system may or may not embody an OA. Thus, given the goal of realizing an OA strategy, together with the use of evolving OSS components and open APIs, *how should program acquisition, system requirements, OAs, and post-deployment system support be aligned to achieve this goal*? As such, this is the central research problem we investigate in this Report in order to identify principles, best practices, and knowledge for how best to insure the success of the OA strategy when OSS and open APIs are required or otherwise employed. Without such knowledge, program acquisition managers and PEOs are unlikely to acquire software-intensive systems that will result in an OA that is clean, robust, and transparent. This may frustrate the ability of program managers or PEOs to realize faster, better, and cheaper software acquisition, development, and post-deployment support.

1.1.2. Issues for Acquisition Research

Based on current research into the acquisition of OA systems with OSS components [Scacchi and Alspaugh 2008, Alspaugh, Asuncion, and Scacchi 2009a,b], our research project through this Final Report seeks to explore and answer the following kinds of research questions: How does the interaction of requirements and architectures for OA systems incorporating OSS components facilitate or inhibit acquisition practices over time? What are the best available ways and means for continuously analyzing the functionality, correctness, and openness of OA when OSS components subject to different, heterogeneous intellectual property (IP) licenses are employed? How do OA systems evolve over time when incorporating continuously improving OSS components? How can use of continuously evolving OSS in OA be combined with the need to formally model, analyze, and manage their evolution? How do reliability and predictability trade-off against the cost and flexibility of an OA system when incorporating OSS components? How should OA software systems be developed and deployed to support warfighter modification in the field or participation in post-deployment system support, when OSS components are employed? In order to examine such acquisition research questions, we focused our research studies in 2009–10 to examine the following topics and activities:

- Investigate the interactions between software system acquisition guidelines, software system requirements, requirements for OSS, and consequences of alternative software system architectures that incorporate different mixes of OSS components, open APIs, and open standards [Scacchi and Alspaugh 2008, Alspaugh, Asuncion, and Scacchi 2009a,b,c]. This entails exploring the balance between development, verification of IP rights, and contractual obligations within continuously improving OSS system elements while managing the evolution of OA systems at design-time, build-time, and release and run-time. Results from investigation of this topic appear throughout Chapters 2 through 6 in this Final Report.

- Develop formal foundations for establishing acquisition guidelines for use by program managers seeking to provide software-intensive systems that realize an OA using OSS technology and processes [Alspaugh, Asuncion, and Scacchi 2009b]. Results from investigation of this topic appear throughout Chapters 2 and 4 in this Final Report.

- Develop the design of a comprehensive automated system that can support acquisition of OA systems so as to determine their conformance to acquisition guidelines/policies, contracts, and related license management issues [Alspaugh, Asuncion, and Scacchi 2009c,

Asuncion 2009]. Results from investigation of this topic are highlighted in Chapter 4 in this Final Report, and appear in greater detail elsewhere [Asuncion 2009].

- Document the investigation, foundations, and results of the research and present final results at the 2010 Acquisition Research Conferences as well as in related research venues, publications, and regular research progress reports. Results from investigation of this topic appear in particular in Alspaugh, Asuncion, and Scacchi [2010], as well as throughout Chapter 2 through 6 in this Final Report.

1.1.3. Prospects for longer-term Acquisition-related research

Each of the U.S. military Services has committed to orienting their major system acquisition programs around the adoption of an OA strategy. Such a strategy embraces and encourages the adoption, development, use, and evolution of OSS. Thus, it would seem there is a significant need for sustained research that investigates the interplay and inter-relationships between (a) current/emerging guidelines for the acquisition of software-intensive systems within the DoD community (including contract management and software development issues) and (b) how software systems that employ an OA incorporating OSS products and production processes are essential to improving the effectiveness of future software-intensive program acquisition efforts. Our research studies have been, and continue to be, focused to address such longer-term acquisition research challenges.

1.2. Acquisition Research Results in this Report

In this Final Report, we present results from our continuing studies that build on the results we have recently produced, that build from our research results presented at the 2010 Acquisition Research Symposium [Alspaugh, Asuncion, and Scacchi 2010][1]. In particular, we highlight two broad kinds of contributions. First, we have taken our previous results spread across multiple research papers, and have now integrated them into a single, coherent report presented in Chapter 2 that has been published in the *Journal of the Association for Information Systems* (JAIS), one of

[1] Our 2010 ARS paper [Alspaugh, Asuncion, and Scacchi 2010], is not included in this report because it has already been delivered for distribution by the NPS ARP. This Final Report instead focuses on research results that build on our 2010 ARS paper. However, both our 2010 ARS paper and its associated presentation materials can be found at http://www.acquisitionresearch.net/cms/_files/FY2010/NPS-AM-10-046.pdf and http://www.acquisitionresearch.net/cms/_files/FY2010/NPS-AM-10-080.pdf, respectively.

the top scholarly research journals in the Information Systems area. The original and unique contribution of this work is the identification, formalization, and demonstration of the issues involved in acquiring a complex system of systems, where each sub-system or component is subject to different contractual obligations and rights conveyed through one or more heterogeneous property rights licenses. We believe our approach is the first such effort to accomplish this result. This is significant both as a formal research result, as well as one that addresses and poses a workable solution for application during actual system acquisition efforts.

Second, we include another paper in Chapter 3 which further builds on the concepts and methods in the preceding paper by extending our efforts to more broadly account for contractual or "supply chain" relationships between system producers, integrators, and consumers/customers (end-user organizations) as a software ecosystem. Once again, our formulation is original and unique, yet practical, in addressing problems of mapping and accounting for how property rights and obligations are transferred from system producers to customers, but mediated by the actions and architectural choices made by system integrators. An earlier version of this second paper [Alspaugh, Asuncion, and Scacchi 2010] was presented at the 2010 Acquisition Research Symposium in Monterey, CA in May 2010. But this version is now in second review for another journal (the Journal of Systems and Software), so we believe it too represents a significant and original research result from our 2009–11 acquisition research efforts.

Third, we present in Chapter 4 a research conference paper from the Fall 2009 [Alspaugh, Asuncion, and Scacchi 2009c] that presents some of the key concepts that were foundational to the two journal papers just described.

Last, are two research papers that further describe the open source software foundations that underlie our studies for open architecture software systems. The first of these (Chapter 5) was invited for presentation at the NSF Workshop on the Future of Software Engineering Research, held in Fall 2010 in conjunction with the Foundations of Software Engineering Conference. The second (Chapter 6) was invited for inclusion in the Encyclopedia of Software Engineering, which was publicly released for publication and distribution also in the Fall of 2010.

1.3. Research Going Forward

Given these five research papers, we have been fortunate to receive follow-on research funding for

this line of acquisition research through a new grant, N0024-10-1-0077 during 2010–11 to further develop and refine the ideas, concepts, and prototyping tools and techniques we are exploring as part of our effort in Acquisition Research supported by the Acquisition Research Program at the Naval Postgraduate School. This new effort continues to build on both our prior acquisition research in general, and specifically on the five research papers presented in this Final Report. A key contribution of this new research effort will be in how we extend our approach to address software product lines and secure software systems within open architecture systems, as noted in Chapter 7 of this Final Report.

We welcome any comments or questions regarding either the results included in this report, or about our current work in progress. We also welcome any opportunity to present or discuss this work with any potential customer enterprise or acquisition unit within a Program Executive Office (PEO) within the DoD, or any of its Services.

1.4. References

Alspaugh, T.A, Asuncion, H., and Scacchi, W. (2009a). Software Licenses, Open Source Components, and Open Architectures, *Proc. 6th Annual Acquisition Research Symposium*, Monterey, CA.

Alspaugh, T.A, Asuncion, H., and Scacchi, W. (2009b). Analyzing Software Licenses in Open Architecture Software Systems, *Proc. Workshop on Emerging Trends in FLOSS Research and Development, Intern. Conf. Software Engineering*, Vancouver, Canada, May.

Alspaugh, T.A, Asuncion, H., and Scacchi, W. (2009c). Intellectual Property Rights Requirements for Heterogeneously Licensed Systems, in *Proc. 17th. Intern. Conf. Requirements Engineering (RE09*), Atlanta, GA , 24–33, September 2009.

Alspaugh, T.A, Asuncion, H., and Scacchi, W. (2010). The Challenge of Heterogeneously Licensed Systems in Open Architecture Software Ecosystems, *Proc. 7th Annual Acquisition Research Symposium*, Monterey, CA.

Asuncion, H. (2009). *Architecture Centric Traceability for Stakeholders (ACTS)*. Doctoral Dissertation, Information and Computer Science, University of California, Irvine, Irvine, CA, July 2009.

Naegle, B. and Petross, D. (2007). Software Architecture: Managing Design for Achieving Warfighter Capability, *Proc. 5th Annual Acquisition Research Symposium*, NPS-AM07104, Naval Postgraduate School, Monterey, CA, May.

Scacchi, W., (2007). Free/Open Source Software Development: Recent Research Results and Methods, in M. Zelkowitz (Ed.), *Advances in Computers*, 69, 243–295, 2007.

Scacchi, W., (2009). Understanding Requirements for Open Source Software, in K, Lyytinen, P. Loucopoulos, J. Mylopoulos, and W. Robinson (Eds.), *Design Requirements Engineering: A TenYear Perspective*, LNBIP 14, Springer Verlag, 467–494, 2009.

Scacchi, W. and Alspaugh, T., (2008). Emerging Issues in the Acquisition of Open Source Software within the U.S. Department of Defense, *Proc. 5th Annual Acquisition Research Symposium*, NPS-AM08036, Naval Postgraduate School, Monterey, CA, May.

THIS PAGE INTENTIONALLY LEFT BLANK

2. Software Licenses in Context: The Challenge of Heterogeneously-Licensed Systems

Thomas A. Alspaugh,	Walt Scacchi	Hazeline U. Asuncion
Georgetown University	University of California, Irvine	University of Washington, Bothell
thomas.alspaugh@acm.org	wscacchi@ics.uci.edu	hazeline@u.washington.edu

Abstract

The prevailing approach to free/open source software and licenses has been that each system is developed, distributed, and used under the terms of a single license. But it is increasingly common for information systems and other software to be composed with components from a variety of sources, and with a diversity of licenses. This may result in possible license conflicts and organizational liability for failure to fulfill license obligations. Research and practice to date have not kept up with this sea-change in software licensing arising from free/open source software development. System consumers and users consequently rely on *ad hoc* heuristics (or costly legal advice) to determine which license rights and obligations are in effect, often with less than optimal results; consulting services are offered to identify unknowing unauthorized use of licensed software in information systems and researchers have shown how the choice of a (single) specific license for a product affects project success and system adoption. Legal scholars have examined how pairs of software licenses conflict but only in simple contexts. We present an approach for understanding and modeling software licenses, as well as for analyzing conflicts among groups of licenses in realistic system contexts, and for guiding the acquisition, integration, or development of systems with free/open source components in such an environment. This work is based on an empirical analysis of representative software licenses and of heterogeneously-licensed systems. Our approach provides guidance for achieving a "best-of-breed" component strategy while obtaining desired license rights in exchange for acceptable obligations.

Published as: T. A. Alspaugh, W. Scacchi, and H.A. Asuncion, Software Licenses in Context: The Challenge of Heterogeneously Licensed Systems, *Journal of the Association for Information Systems*, 11(11), 730–755, November 2010.

2.1. Introduction

The hallmark of Free/Open Source Software (FOSS) is that the *source code* is available for remote access, open to study and modification, and available for redistribution to others with few constraints, except the rights and obligations that insure these freedoms. FOSS sometimes adds or removes similar freedoms or copyright privileges depending on which FOSS copyright and end-user license agreement is associated with a particular FOSS code base [Fontana 2008, Rosen 2005]. Some licenses for "free software" such as the group of GNU General Public License (GPL) and Lesser General Public License (LGPL) versions (Free Software Foundation 1991, 1999, 2007a, 2007b, 2007c) are well known and widely used, while others are obscure and sometimes specific to a particular software vendor. The Open Source Initiative (OSI 2009), whose Web portal gives information on many facets of FOSS, especially FOSS licenses, currently certifies more than 50 FOSS licenses, thus indicating a diverse ecology of freedoms, copying rights, and other license obligations and constraints.

Some FOSS licenses overlap or subsume one another's rights, while others present potential conflicts when comparing one license to another. Consequently, FOSS developers generally choose a single license to apply to their FOSS project as part of their governance regime (de Laat 2007). The choice of FOSS license to apply has been a defining characteristic of most FOSS projects, where the license chosen may connote not only an intellectual property sharing regime, but also a statement about beliefs, values, and norms expected to be shared by FOSS project developers, as well as affiliation within a larger social movement (Elliot and Scacchi 2003, 2008, Roberts et al. 2006). However, a single license may not be sufficient to provide "copyleft" access to non-software specific data objects, representations, processing rules, or visual renderings. Similarly, with ever more FOSS components becoming available with different FOSS licenses, and some now even offered under multiple licenses[2] or recognizing that different versions of a given software component may have different licenses or license constraints, then software and information system developers face a growing challenge: to determine how multiple software licenses interact, whether during system design (i.e. at "design time"), while compiling and linking source code to produce an executable program/binary (at "build time"), or when installing and running a newly acquired/downloaded version of software from a FOSS project or other provider

[2] MySQL (2006) is offered under a dual-license choice of GPL (any version) or a proprietary license, and Mozilla now offers a tri-license choice of the Mozilla Public License (MPL) version 1.1 or later, GNU General Public License version 2.0 or later, or GNU Lesser General Public License version 2.1 or later, for its core software offerings (Mozilla 2009).

that may need to interoperate with other software programs (at “run time”).

The problem we address in this paper is to understand and analyze what happens when software systems are developed from FOSS or proprietary components that are not all under the same license. What license applies to the resulting system? What rights or obligations apply? How can one determine which license constraints match, subsume, or conflict with one another? We refer to this problem as *the challenge of heterogeneously-licensed systems* (HLSs), and we find that a growing number of firms and government agencies must increasingly address this challenge. Consider the following two examples: one short and the second more detailed.

First, when the Bank of America took over operation of the Merrill Lynch (ML) trading firm in 2008, ML was known as a leading developer of in-house financial and trading systems incorporating FOSS components. However, the Bank now had to rapidly determine whether this corporate takeover constituted a “software distribution” by ML as well as what consequences might arise from integrating the ML systems with its own (Asay 2008), because this would likely create an HLS.

Second, Unity3D is an interactive environment for modeling and animating 3D graphic objects and object compositions within computer games or virtual worlds. Unity3D is an HLS, as seen in its software copyright license agreement that lists 18 externally produced components or groups of components, apparently distributed under eleven distinct licenses (Unity Technologies 2009). So which “license” rights and obligations apply to this software? Is it the concatenation, union, or intersection of the license constraints found in each of the identified license copyrights? Do any of these license constraints conflict with one another (e.g. stipulating no redistribution of software, versus ensuring the right to software redistribution)? How does the architectural configuration of the software components associated with a given license affect which component licenses interact and which do not? Understanding the rights and obligations incorporated into the Unity3D system license requires understanding and analyzing the corresponding terms and conditions of each of these licenses and potentially understanding the architecture in which Unity3D incorporates them. This is the burden facing software consumers, and it appears to be one that is growing. It is also a burden facing software integrators, because they must ensure that they can give appropriate rights to (and impose acceptable obligations on) their consumers.

The current state of the art of research in the fields of law and FOSS does not address these concerns, as we will show in more detail in the Related Research section. Legal researchers have

examined interactions between pairs of FOSS licenses in the abstract, but not in the context of real systems and the architectural components and configurations found there. FOSS research in this area has concentrated on recovering or confirming the license of an individual homogeneously-licensed software component, with only one group (other than ourselves) examining the application of FOSS licenses in the context of actual HLSs, and that only of pairs of licenses applied in a small fixed number of architectural contexts. None of these approaches provide answers to the questions posed above.

We believe these answers can be determined in part through systematic empirical means that rely on analysis of (a) the interpretation of software license terms and conditions within different FOSS licenses, along with (b) the configuration of software components (and licenses) that denote the architectural composition of the resulting system.[3] In particular, we argue that (b) requires an open source model of the architectural configuration of a system in which each component has an open (published) interface. We call such a configurational model an *open architecture* (OA) (Oreizy 2000). In current practice, an OA is not available to external developers or users when proprietary software components/systems are included, as is the case for Unity3D. We find that the size and complexity of the HLSs we have examined, combined with increasingly large numbers of licenses involved, outstrip the ability of developers to manually determine the rights and obligations for the system and to identify potential and actual conflicts. Our research has thus been drawn to models and approaches that support automatic analysis, calculations, and guidance for licensing challenges, and that can be integrated into the processes and tools that HLS developers already use.

Our efforts are directed at theory building because there is no existing theory for understanding HLSs and analyzing them to determine overall license rights, obligations, and conflicts. As such, we are developing a theory-based, empirically verified model as an operational theory for how to understand and analyze FOSS licenses in ways that determine where HLS architectures have conflicting rights and obligations, as well as how these architectures may be modified to remove the conflicts. Because existing theory, as outlined under Related Research, is not sufficient to explain or analyze the license interactions through system architecture that arises in HLSs, we extend it through a *design science* approach (Hevner et al. 2004), extending the existing "kernel" theories through a grounded-theory qualitative analysis of representative licenses to identify

[3] Our efforts are not intended to be construed as offering legal advice, but our approach is adaptable to different legal framings or interpretations, which are beyond the scope of this paper. Thus, our approach does not claim to offer complete or definitive answers according to an existing legal interpretation.

missing constructs (Glaser and Strauss 1967) and validating the extended theory by embodying it in automated development tools and applying it in current-day contexts. The tools provide a proof of concept and a starting point for a production-quality capability for analyzing, guiding, and verifying license governance and compliance of HLSs and are discussed elsewhere (Alspaugh et al. 2009a). This capability supports the needs of large enterprises that increasingly develop complex information systems using or reusing existing FOSS systems/components, rather than developing each system from a blank slate.

The goal of this work is to develop an extensible, automatable theory for analyzing the obligations and rights of software licenses in the kinds of complex systems now seen in enterprises. The theory is based on the legal foundations of licenses and contracts, and accounts for the range of current and future license provisions. It makes it a practical possibility to produce systems of best-of-breed components, whether FOSS or proprietary and under whatever licenses, while addressing not only the classical requirements of functionality, reliability, testability, etc., but also the licensing-oriented requirements that arise for HLSs. These include production of a system that can be legally used, legally distributed, legally modified and evolved to meet changing requirements, and for which it can be reliably shown what rights are obtainable and what obligations must be met in order to obtain specific desired rights.

Our approach starts by constructing a semantic model of the rights and obligations found in the text of FOSS licenses. This modeling need only be done once for a license, unless or until the license or its interpretation is changed. Next, the analytical methods we propose determine whether conflicts arise when specific FOSS-licensed components are incorporated within an OA, working from component attributes that give the semantic models for their corresponding license. Last, our testbed environment demonstrates that our theory can be supported with automated tools, which offer the potential to analyze large, complex HLSs at design time, build time, or run time. As a result, our theory is operational in that it can be applied to a complex HLS to empirically determine whether its architecture possesses conflicting rights or obligations, given its components' licenses and interconnections. Finally, our theory is not limited to a single pre-determined interpretation of the meaning of terms for rights and obligations found in FOSS licenses. It allows for alternative legal interpretations of a license, which can then be analyzed and supported by our approach and with our automated tools, when so coded to reflect these interpretations.

Our theory provides answers to the questions posed above. By examining HLSs, we find that it is

typical that no license, *per se*, applies to the resulting system. Instead, there is a collection of rights available for the system, and for each right there are corresponding obligations that must be met. We find that the collection of rights may be empty, and that if specific rights are desired, such as the right to use the system and the right to distribute it, these rights must be considered at each stage of development from design through distribution. The specific rights and obligations are determined by the licenses of the system's components and by the architectural configuration in which they are combined. Informal analysis of license constraints by experts can identify which ones may match, subsume, or conflict, but our theory supports a formal analysis that can be automated for non-experts. These issues are addressed in our other work, which implements and applies an initial version of the theory presented and elaborated here (Alspaugh et al. 2009b).

The remainder of the paper is organized as follows. In the next section, “A Motivating Example”, we present a motivating example displaying some of the non-obvious ways in which FOSS license conflicts can manifest themselves. The “Related Research” section reviews prior research that helps inform our understanding of the problem we are addressing. Emphasis is directed to studies that rely on empirically-grounded investigations of FOSS development projects and outcomes. The “Intellectual Property (IP) Basics” section summarizes IP law, and “Software Licensing and FOSS Licenses” does the same for licenses.

The section “A Theory of Software Licenses and their Application” summarizes existing legal theory and presents the basic meta-model and reasoning rules that can be derived from it. Following this, we present an empirical study of a selection of widely-used software licenses, and a study approach that can be extended to additional licenses, and use the results to extend the meta-model and render it strong enough to support automated application of licenses and calculation involving them. In this regard, we seek to provide a semantic or logical model (a set of logical constraints) for specifying license rights and obligations, such that any use of a given license reuses its logical model.

The next section “License Architecture of Heterogeneously-Licensed Systems” presents the types of architectural components and connectors that our study showed as significant for FOSS licensing. Following this, “Embodying and Applying the Models and Analysis” presents an external validation of our theoretical results, in which we embodied the theory in an interactive software architecture environment and applied it to HLSs. Such an environment demonstrates the potential to scale up the analysis to large, complex systems that may be too difficult to analyze without such

automated support. The value of the environment is also demonstrated through detection of conflicting licenses constraints across components that can be resolved through changes to the system's architectural configuration, such as through the introduction of license firewalls or substitution of similar software components with different licenses. This section is then followed by the “Discussion” section that offers our view of any HLS as being subject to a “virtual license” that results from an empirical analysis of the different licenses superimposed on the architecture of a composed software system. Such a virtual license represents a union of the license constraints across the components once conflicting license constraints have been resolved and an intersection of the license rights that result. We then conclude our study with a final statement of the results we have accomplished through our approach.

2.2. A Motivating Example

Shortly before beginning the research described here, we prototyped a new multimedia content management portal that included support for videoconferencing and video recording and publishing. Our prototype was based on an existing Adobe Flash Media Server (FMS), for which we developed both broadcast and multi-cast clients for video and audio that shared their data streams through the FMS. FMS is a closed source media server for which the number of concurrent client connections is limited, with the limit determined by the license fee paid. We could invite remote users to try out our clients and media services (up to the connection limit), but because the FMS license did not allow redistribution, we found we could not offer interested users a copy of the run-time environment that included the FMS. We could distribute everything but the FMS though our compiled components were built to use our copy of the FMS. Consequently, recipients would be unable to run the system even if they purchased their own FMS license. The only useful way to distribute our portal system was in the form of the source code of our locally-developed clients and services. A potential user would need to license, download, and install his own copy of the FMS, configure our source code to use it, and rebuild our system. In our view, this created a barrier to sharing the emerging results of our prototyping effort.

We subsequently undertook to replace the FMS with Red5, an open source Flash media server, so that we could distribute a complete run-time version of our content management portal to remote developers. Now these developers could install and use our run-time system as-is, or if they preferred, they could download the source code, revise, build, and configure it to suit their own needs and share their own run-time version.

Our experience illustrates how heterogeneous licensing can interact with system architecture decisions to hamper common software research and development activities in surprising ways even for experienced FOSS designers and developers, and if not planned for successfully, can cause substantial unexpected costs and delays. Our license theory, embodied in the tool described later in the paper, would have highlighted the lack of distribution rights and modeled the non-substitutability of the FMS server at system design time rather than much later at system distribution and run time.

2.3. Related Research

It has been typical until recently that each software or information system is designed, built, and distributed under the terms of a single proprietary or FOSS license, with all of its components homogeneously covered by that same license. The system is distributed, with sources or executables bearing copyright and license notices, and the license gives specific rights while imposing corresponding obligations that system consumers (whether external developers or users) must honor, subject to the provisions of contract and commercial law. Consequently, there has been some very interesting study of the choice of FOSS license for use in a FOSS development project and its consequences in determining the likely success of such a project.

Brown and Booch (2002) discuss issues that arise in the reuse of FOSS components, such as that interdependence (via component interconnection at design time, or linkage at build time or run time) causes functional changes to propagate and versions of the components evolve asynchronously, giving rise to co-evolution of interrelated code in the FOSS-based systems. If the components evolve, the OA system itself is evolving. The evolution can also include changes to the licenses, and the licenses can change from component version to version.

Stewart et al. (2006) conducted an empirical study to examine whether license choice is related to FOSS project success, finding a positive association following from the selection of business-friendly licenses. Sen, Subramaniam, and Nelson in a series of studies (Sen 2007; Sen et al. 2008; Subramaniam et al. 2009) similarly find positive relationships between the choice of a FOSS license and the likelihood of both successful FOSS development and adoption of corresponding FOSS systems within enterprises. These studies direct attention to FOSS projects that adopt and identify their development efforts through use of a single FOSS license. However, there has been

little explicit guidance on how best to develop, deploy, and sustain complex software systems when heterogeneously-licensed components are involved, and thus, multiple FOSS and proprietary licenses may be involved.

Legal scholars have examined FOSS licenses and how they interact in the legal domain but not in the context of HLSs. St. Laurent (2004) examines twelve FOSS licenses, including several no longer in wide use, and compares them to a hypothetical proprietary license he created; license interactions and conflicts are only very briefly discussed, and only in general terms. Rosen (2005) surveys eight FOSS licenses and creates two new ones written to professional legal standards. He examines interactions primarily in terms of the general categories of reciprocal and non-reciprocal licenses, rather than in terms of specific licenses. Rosen was general counsel for the Open Source Initiative (OSI 2009). Fontana et al. (2008) primarily focused on guidance for FOSS projects on a number of legal issues, but provide a good and authoritative survey of FOSS licenses, especially the GPL group of licenses. Fontana et al. are lawyers (with one exception) associated with the Software Freedom Law Center; Fontana and Moglen were two of the authors of the GPL, LGPL, and AGPL version 3 licenses. Finally, Kemp (2009) reviews significant court cases that have been pursued and how they do or do not address issues concerning the propagation of rights and obligations across programs depending on how they are derived, contained, compiled, or linked at build time, especially when the GPLv2 license is involved. Common to this legal scholarship is an approach that analyzes licenses and the interactions among them abstractly rather than in the context of an HLS and on at most a pairwise basis. The characteristics of the software in which the licenses interact are not taken into account, or at most in very general terms, even though almost every FOSS license is framed in terms of the software and architectural constructs in existence when the license was written.

Ven and Mannaert (2008) discuss the challenges faced by independent software vendors developing an HLS. They focus on the evolution and maintenance of modified FOSS components. Tuunanen et al. (2009) exemplify most work to date on HLSs, in that they focus on reverse engineering and recovery of individual component licenses on existing systems rather than on guiding HLS design to achieve and verify desired license outcomes. Their approach does not support the calculation of HLS virtual licenses. Many more researchers have worked from this after-the-fact point of view (Gobeille 2008, Di Penta et al. 2010).

German and Hassan (2009) describe a license as a set of grants, each of which has a set of

conjoined conditions necessary for the grant to be given. Figure 1 is a meta-model equivalent to their notational license definition. They analyze interactions between pairs of licenses in the context of five types of component connection. They also identify twelve patterns for avoiding license mismatches found in an empirical study of a large group of FOSS projects and characterize the patterns using their model. Their license models, developed independently from our work at about the same time, are equivalent to the first level of our license models and provide confirmation that our work builds on accepted foundations. As we show below, our license model goes beyond German and Hassan's to address semantic connections between obligations and rights that existing FOSS licenses exhibit, and to connect with the structure of software systems in a general and extensible way rather than a fixed set of cases.

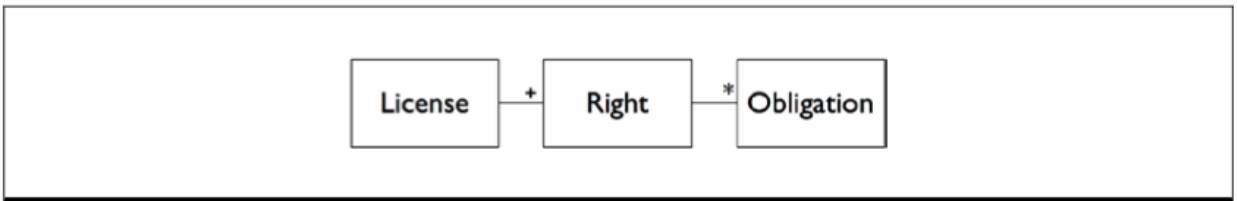


Figure 1. The meta-model used by German and Hassan (2009)

Other previous work examined how best to align acquisition, system requirements, architectures, and FOSS components across different software license regimes to achieve the goal of combining FOSS with proprietary software that provide open APIs when developing a composite “system of systems” (Scacchi and Alspaugh 2008). This is particularly an issue for the U.S. Federal Government in its acquisition of complex software systems subject to Federal Acquisition Regulations (FARs) and military Service-specific regulations. HLSs give rise to new functional and non-functional requirements that further constrain what kinds of systems can be built and deployed as well as recognize that acquisition policies can effectively exclude certain OA configurations, while accommodating others, based on how different licensed components may be interconnected.

2.4. Intellectual Property (IP) Basics

Software licenses are primarily concerned with copyright. Copyright is defined by Title 17 of the U.S. Code and by similar law in many other countries. It grants exclusive rights to the author of an original work in any tangible means of expression, namely the rights to reproduce the copyrighted work, distribute copies, prepare derivative works, distribute copies of derivative works, and (for certain kinds of work) perform or display it. Because the rights are exclusive, the author can prevent others from exercising them, except as allowed by “fair use”. The author can also grant

others any or all of the rights or any part of them; one of the functions of a software license is to grant such rights and define the conditions under which they are granted.

2.5. Software Licensing and FOSS Licenses

Traditional proprietary licenses typically grant a minimum of rights that licensees need to use the licensed software, retaining control of software the licensor has produced and restricting the access and rights outsiders can have. In contrast, FOSS licenses are designed to encourage sharing and reuse of software and typically grant as many rights as possible consistent with that goal. FOSS licenses are conventionally classified as academic or reciprocal. An *academic* FOSS license such as the Berkeley Software Distribution (BSD) license, MIT license, or Apache Software License grants nearly every copyright right for components and their source code, and imposes few obligations. Anyone can use the software, create derivative works from it, or include it in proprietary projects. Typical academic obligations are simply to not remove the copyright and license notices (University of California 1998).

Reciprocal or *copyleft* FOSS licenses take a more active stance towards the sharing and reuse of software by imposing an obligation that reciprocally-licensed software only be combined (for various definitions of “combined”) with software that is in turn also released under the reciprocal license. The primary goal of reciprocity in licenses is to increase the amount of FOSS by encouraging developers to bring additional software into the FOSS commons and to prevent improvements to OSS software from vanishing behind proprietary licenses. Example reciprocal licenses are the GPL licenses, the Mozilla Public License (MPL), and the Common Public License (CPL).

Common practice has been for a FOSS project to choose a single license under which all of its distributions are released and to require developers to contribute their work under conditions compatible with that license. For example, the Apache Contributor License Agreement (Apache 2009) grants enough of each contributor's rights as an author to the Apache Software Foundation for it to license the resulting systems uniformly under the Apache license. This sort of rights regime, in which the rights to a system's components are homogeneously granted and the system has a single well-defined OSS license, was universal in the early days of FOSS and continues to be widely practiced.

2.6. A Theory of Software Licenses and their Application

2.6.1. Legal Grounding

Under U.S. law and the law of most countries, a license can be either a bare license or a contract license. A *bare license* simply grants one or more copyright or patent rights from the copyright and/or patent holder to another person. A *contract license* is constructed in the form of a contract, involving an exchange of promises between the parties. In a contract license, the licensor grants one or more rights in exchange for some consideration from the licensee receiving them. The consideration given by the licensee may be very small, as little as “a peppercorn” in the traditional explanation, but it cannot be nothing. However, in addition to a payment or other obvious consideration, it can take the form of “(a) an act other than a promise, or (b) a forbearance, or (c) the creation, modification, or destruction of a legal relation” (American Law Institute 1981). In FOSS licensing the fundamental consideration is typically interpreted as the licensee's “detrimental reliance” on the licensed rights, or in other words the reliance on the software that would be to the licensee's detriment if the software were withdrawn. A license, whether bare or contract, can also impose specific non-consideration obligations as a condition of the license grant. Most FOSS licenses are drawn as contract licenses in order to benefit from the well established case law on interpretation of contract provisions, with the exception of the GPL family of licenses which are drawn as bare licenses (Rosen 2005, Gordon 1989, Determann 2006, Guadamuz 2009, Hillman and O'Rourke 2009, Kemp 2009).

In a seminal article published in 1913 and cited up to the present day, Hohfeld presented a theory by which he proposed to resolve the imprecise terminology and ambiguous classifications he found in use for legal relationships (Hohfeld 1913). He set out a system of eight *jural relations* to express and classify all legal relationships between parties. The eight relations are listed below followed by informal descriptions in parentheses. The first four regulate ordinary actions

- *right* (“may”)
- *no-right* (“cannot”)
- *duty* (“must”)
- *privilege* (“need not”)

whereas the second four regulate actions that change legal relationships

- *power* (“may change relation R”)
- *disability* (“cannot change R”)
- *liability* (“must accept changes in R”)
- *immunity* (“need not accept changes in R”)

Each relation has an *opposite* relation whose sense is its opposite, and a *correlative* relation whose sense is its complement. The relations are shown with their opposites and correlatives in Figure 2.

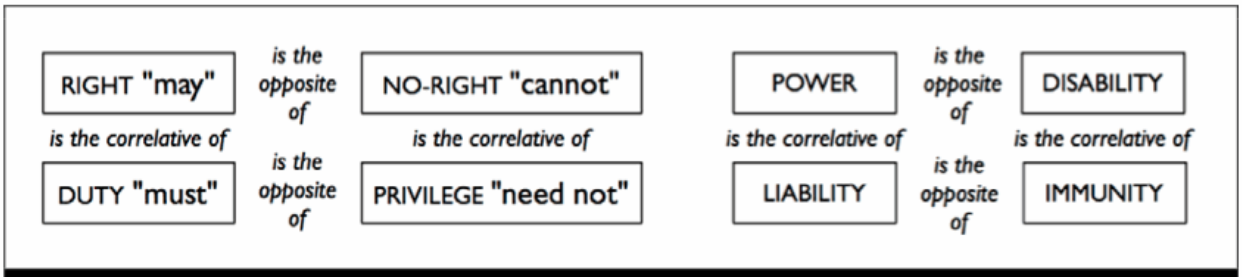


Figure 2. Hohfeld’s jural relations, indicating opposites and correlatives

It has been argued that Hohfeld's jural relations provide a sound basis for legal expression, first by Hohfeld himself and more recently with specifics and supporting data by Allen and Saxon (Allen and Saxon 1995). The relations have also been used by many other researchers in law and other fields, such as Balkin (1991), Gordon (1989), and Humphris-Norman (2009) in law, Daskapulu and Sergot (1997) and McCarty (2002) in artificial intelligence, and Huhns and Singh (1998) and Siena et al. (2008) in software engineering.

2.6.2. A Basic Meta-Model for Licenses

We derived a basic meta-model for software licenses by analyzing the structures defined in law and dividing them into constituent parts, as shown in Figure 3. An enactable right or obligation is composed of the action performed, the actor performing it, and the modality expressing whether the action is allowed, required, or forbidden in terms of Hohfeld relations. The two parties involved in a license are the licensor and licensee. In addition, it is clear that while a right or obligation may involve virtually any action, those actions that are regulated by copyright and patent law are distinguished as of particular importance. Overall, a license is modeled as one or more rights to be granted, each right corresponding to zero or more obligations to be performed in exchange. Both rights and obligations are modeled as tuples of actor, modality, and action, with rights having

modalities “may” or “need not” and obligations “must” or “cannot”.

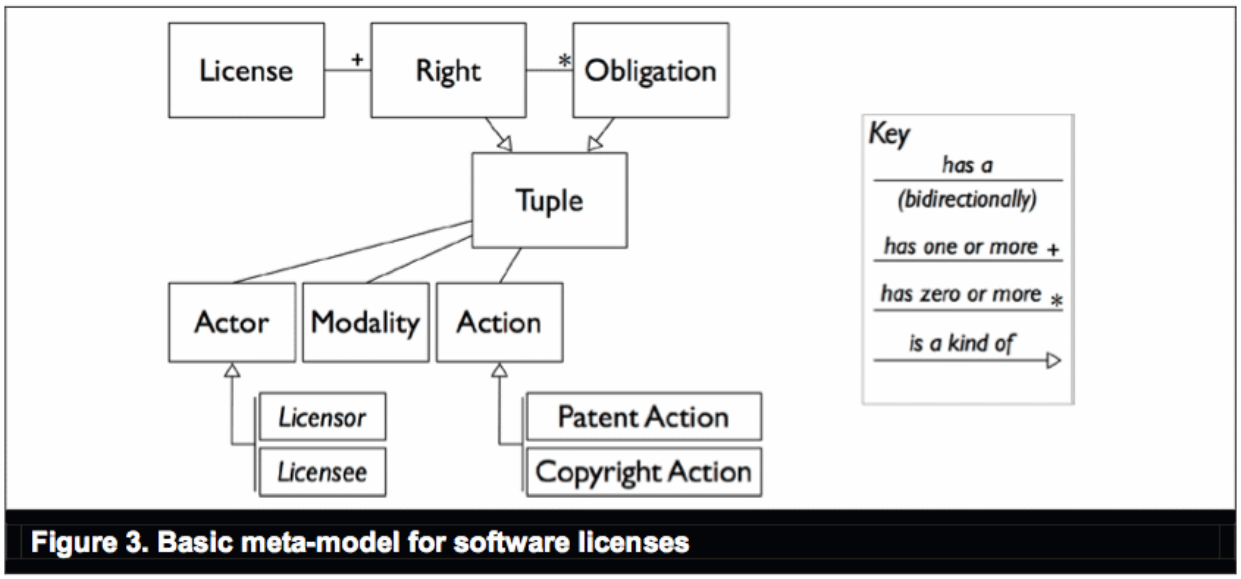


Figure 3. Basic meta-model for software licenses

2.6.3. Reasoning Rules in this Meta-Model

The legal grounding suggests and supports two rules of implication for licenses following this model. The first rule expresses the contractual exchange of promises: a right is granted only if its corresponding obligations are fulfilled. The second rule expresses Hohfeld's correlatives: an obligation can be fulfilled only if its actor has the right to fulfill it. If that right is not immediately available, then the actor may have to obtain it by first fulfilling other obligations. The obligation's correlative is that prerequisite right. If the obligation is that the actor “must” perform the action, then the prerequisite right is that he “may” perform it, while if the obligation is that the actor “cannot” perform it, then the requisite right is that he “need not”.

The chaining of these two rules is sketched in Figure 4, in which the desired right depends (by rule 1) on a necessary obligation, which in turn depends (by rule 2) on the obligation's correlative right as a prerequisite. If the prerequisite right is not immediately available, it must be obtained by performing the obligations necessary for it and so forth until all the obligations are performable.

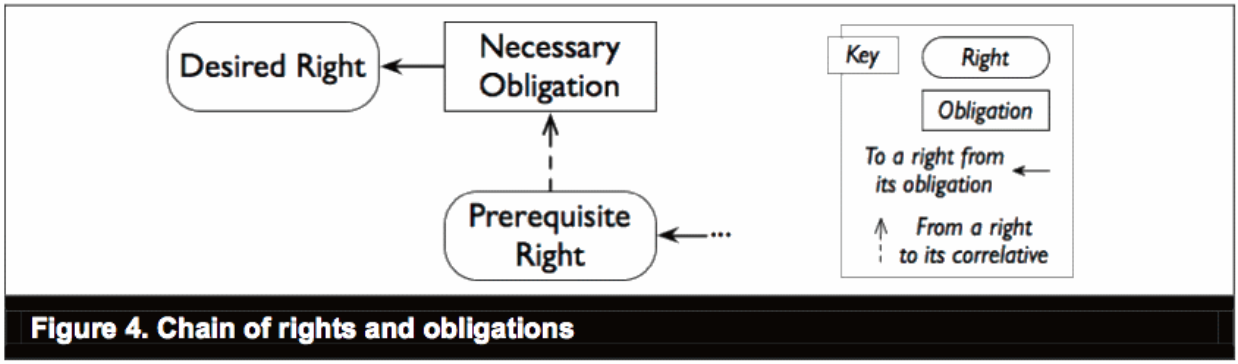


Figure 4. Chain of rights and obligations

This chaining is illustrated most clearly by the reciprocal licenses, in which (in general terms) the right to distribute a derivative work derived from a reciprocally-licensed original carries the obligation to distribute it only under that reciprocal license. The correlative right is the right to distribute copies under that license; if all the originals on which the derivative was based were obtained under the same license, then that right is immediately available, but if the originals were under two or more different licenses, then it may not be immediately possible to distribute the derivative under all of them because their provisions may conflict. If so, the correlative right would have to be obtained by recursively fulfilling whatever obligations (if any) would grant it. It may not be possible to obtain the correlative right at all, either because the relevant licenses do not grant it under any circumstances, or because the necessary obligations are in conflict and cannot be simultaneously fulfilled. In such a case, the desired right is not available and there is no way to obtain it.

2.6.4. Extending the Theory Using Empirical Data

The view and analysis of software licenses as described in the previous section are appropriate as a basis for human reasoning about licenses in the abstract, but are not sufficient for formal analysis and automation. For example, how does GPL version 2.0 apply to a specific software component `code.c`? A human being could (with some effort) list the rights and obligations for `code.c` using that view as a basis, but the process could not be automated using it. If the copyright right to distribute copies of `code.c` were desired, a human being could identify the provisions of GPL version 2.0 that applied, but again this could not be automated.

In addition, it was not clear to us whether the theory of the previous section fully accounted for actual software licenses, or whether (and if so, to what extent) experts working with licenses use experience and shared tacit knowledge to account for features and interactions not covered by

theory.

We conducted a grounded-theory qualitative study of software licenses in order to address these issues. The research questions for the study were:

- Do software licenses match the meta-model in Figure 3?
- What features if any are not accounted for by the meta-model?
- What software-architectural structures are referenced in licenses, and how do they interact with the structures in the meta-model?
- In what ways does the meta-model need to be extended in order to support automated calculation, inference, and application of abstract license provisions to concrete software?

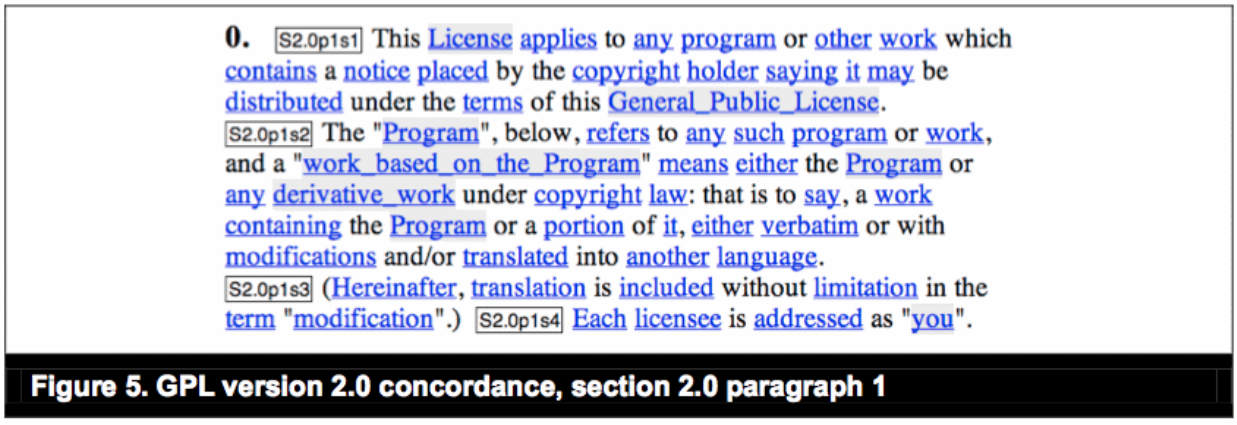
The research questions for the study were developed and elaborated using the Goal Question Metric approach (Basili et al. 1994). The texts of ten licenses were analyzed first by semantic parameterization (Breaux et al. 2008) in order to account for differences in terminology and conceptualization between the studies licenses, and were then chunked and open-coded systematically in accordance with qualitative practice (Creswell 2003, Miles and Huberman 1994). The extensions to the meta-model were generated from the data using a grounded-theory approach (Glaser and Strauss 1967, Corbin and Strauss 2007).

We analyzed these ten licenses:

- Apache 2.0
- Berkeley Software Distribution (BSD)
- Common Public License (CPL)
- Eclipse Public License 1.0 (EPL)
- GNU General Public License 2 (GPL)
- GNU Lesser General Public License 2.1 (LGPL)
- MIT
- Mozilla Public License 1.1 (MPL)
- Open Software License 3.0 (OSL)
- Corel Transactional License (CTL)

The licenses were chosen to represent a variety of kinds of licenses, and include one proprietary (CTL), three academic (Apache, BSD, MIT), and six reciprocal licenses (CPL, EPL, GPL, LGPL, MPL, OSL) that take varying approaches in implementing copyleft. The nine FOSS licenses accounted for approximately 75% of FOSS software at the time of this writing (Black Duck 2009) and include the licenses presented by Rosen as representative (2005). The texts of the FOSS licenses are from the Open Source Initiative web site (OSI 2009), and the text of the proprietary CTL license is from Corel's web site (Corel 2009).

The four stages of the analysis were as follows. First, we performed a word and term level analysis of the texts. We disambiguated the text in several ways. We identified forward and backward references and linked them to their references, identified synonyms, and distinguished polysemes expressing different meanings with identical wording. We identified terms of art such as “Derived Work” from copyright law and specialized terms such as “work based on the Program” for GPL and “Electronic Distribution Mechanism” for MPL that have a license-specific meaning. We then constructed (automatically) a concordance of the resulting text: an index giving each instance of the significant words in the licenses. We excluded minor words such as articles, conjunctions, and prepositions whose use in a particular license carries no specialized meaning. Then we (automatically) tagged each sentence with its section, paragraph, and sentence sequence numbers. Figure 5 shows a portion of the concordance for GPL.



0. S2.0p1s1 This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General_Public_License.
S2.0p1s2 The "Program", below, refers to any such program or work, and a "work_based_on_the_Program" means either the Program or any derivative_work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language.
S2.0p1s3 (Hereinafter, translation is included without limitation in the term "modification".) S2.0p1s4 Each licensee is addressed as "you".

Figure 5. GPL version 2.0 concordance, section 2.0 paragraph 1

Second, we identified parts of each license having no legal force, such as GPL 2's “Preamble” section, or parts that dealt with anything other than copyright, such as patents, trademarks, implied warranty, or liability. We iterated this step using the concordance to confirm our identifications. The remainder of our analysis focused on copyright.

Third, using the concordances across the licenses, and guided by legal work on FOSS licenses (Fontana et al. 2008, Rosen 2005, St. Laurent 2004), we identified words and phrases with the same intentional meaning and textual structures parallel among the licenses. Working from these, we iterated to identify natural language patterns in the licenses and patterns of reference among license provisions and to the licensed software and its context.

Fourth, we chunked and open-coded the text into segments addressing the same issues, with the set of issues arising from the texts themselves. We compared the categories and individual segments of text with the meta-model, noting cases of agreement and cases in which the meta-model did not account for significant features. As the iterative process progressed, new meta-model structures arose from the categories and from the structures expressed in the texts. The iteration converged on an extended meta-model shown in Figure 6 and elaborated in Figure 7 that summarizes what we found as a set of semantic relations. Our analysis confirmed that each license we examined consists of one or more *rights*, each of which entails zero or more *obligations*. Rights and obligations have the same structure, a relation expressed as a tuple comprising an actor (the licensor or licensee), a modality, an action, an object of the action, and possibly a license referred to by the action.

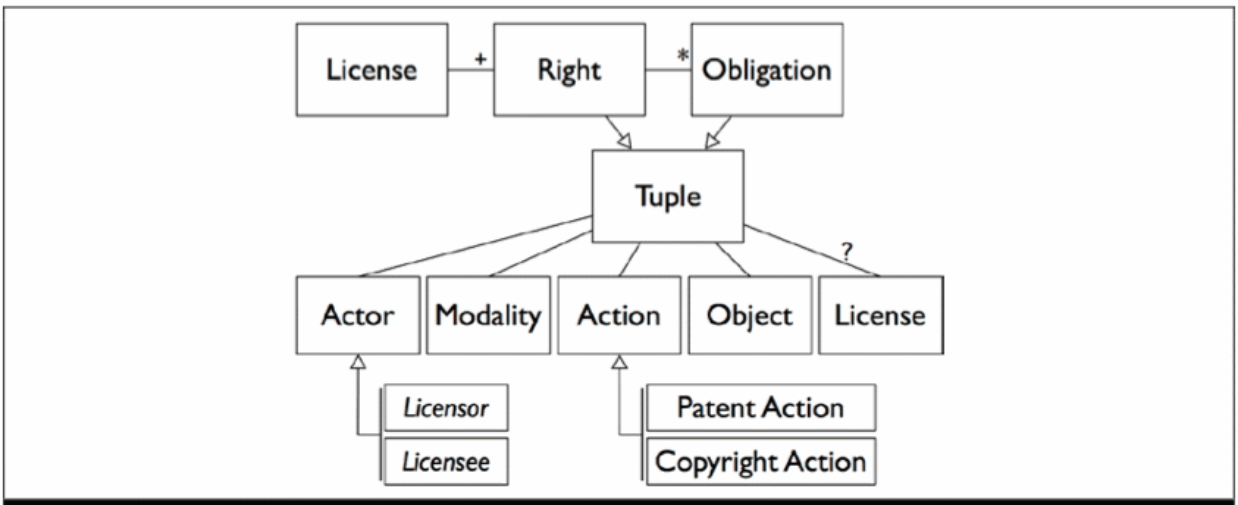


Figure 6. The extended meta-model for licenses, adding Object and License

We found a wide variety of license actions, some defined in copyright law and others defined by specific licenses. Many of the objects within actions are references, and the kinds of references we identified in the ten analyzed licenses are listed in Figure 7 along with the contexts in which they occur. The objects for rights are self-explanatory from their names. If the object of an obligation is *"Right's Object"* then that obligation is instantiated with the same object as its

corresponding right (for example hypothetical right “Licensee may modify X” with obligation “Licensee must publish modifications of X”), *“All Sources of Right's Object”* results in as many obligations as the right's object has source files, with an instance of the obligation for each one, *“X Scope Components”*, for each reciprocal license X, results in as many obligations as there are components within the scope of propagation of X's obligations that includes the right's object, and *“X Scope Sources”* analogously for sources of components in that scope.

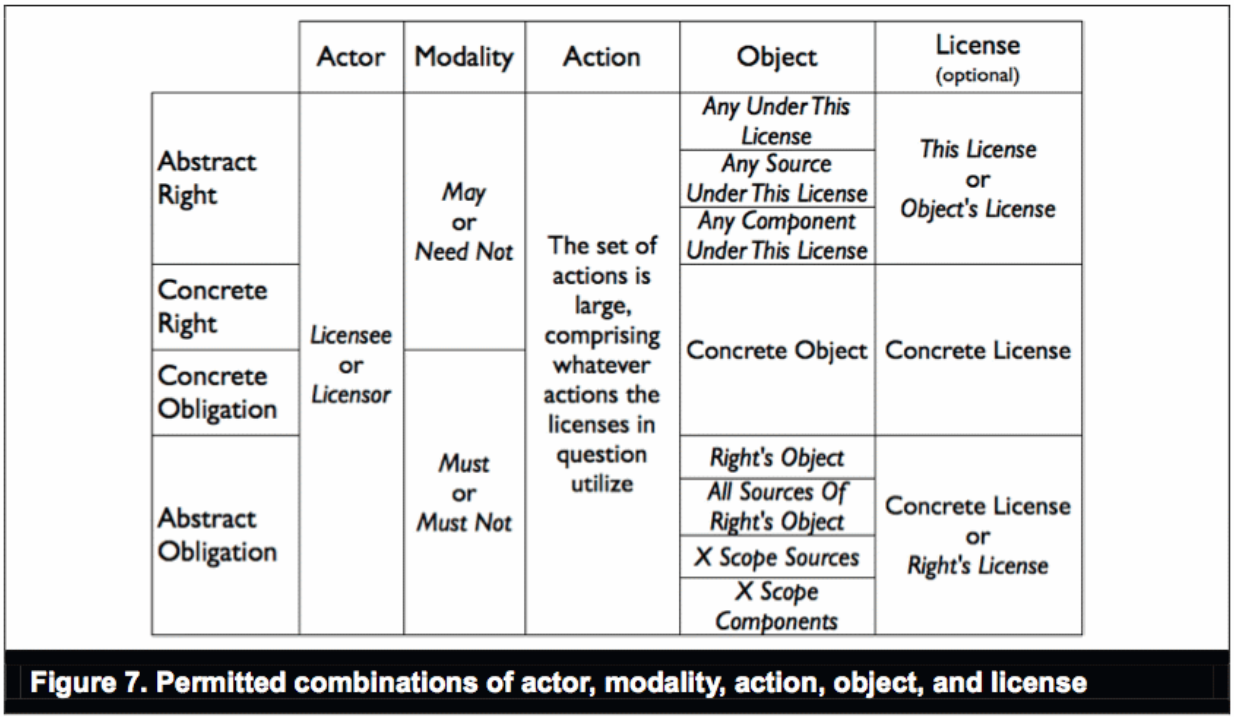


Figure 7. Permitted combinations of actor, modality, action, object, and license

Note that while some objects such as *“Right's Object”* refer only to the same entity that the corresponding right does, others such as *“All Sources of Right's Object”* and *“X Scope Components”* induce more complex patterns. In order to instantiate an abstract right or obligation containing such an object, it is necessary to have information about the system they refer to, such as the way in which components are interconnected, in order to determine each license scope in that system or the sources from which a component was built. We call the abstraction of the system that provides this information the *license architecture* of the system and discuss it in the next section.

As a cross-check on the validity of our analysis, we implemented the license meta-model in Java and expressed the licenses using it, generated a restricted natural language (RNL) form for each license right and obligation, and compared them with the originals. The RNL text for an example *abstract right* (one not bound to a specific object and context) extracted from the BSD license is:

Licensee · may · distribute <*Any Source*> under <*This License*>

where “distribute under” is an action reserved by copyright law and the abstract object <*Any Source*> quantifies the right over all sources licensed under the license containing the right (here, BSD). An example concrete obligation (one bound to a specific context) is:

Licensee · must · retain the [BSD] copyright notice in [file.c]

where “retain the copyright notice” is an action that is not reserved by copyright law, BSD is the concrete license the action references, and file.c is the concrete object the action references. The encoded actions contain tokens identifying where the tuple's object and (if present) license are inserted, for example in the GPL action “sub-license under” which becomes “sub-license *OBJECT* under *LICENSE*” for a particular object and license. Figure 8 informally illustrates how actions may contain concrete objects or licenses, references to objects or licenses bound elsewhere, or quantifiers using the information in the license architecture to produce sets of concrete rights or obligations.

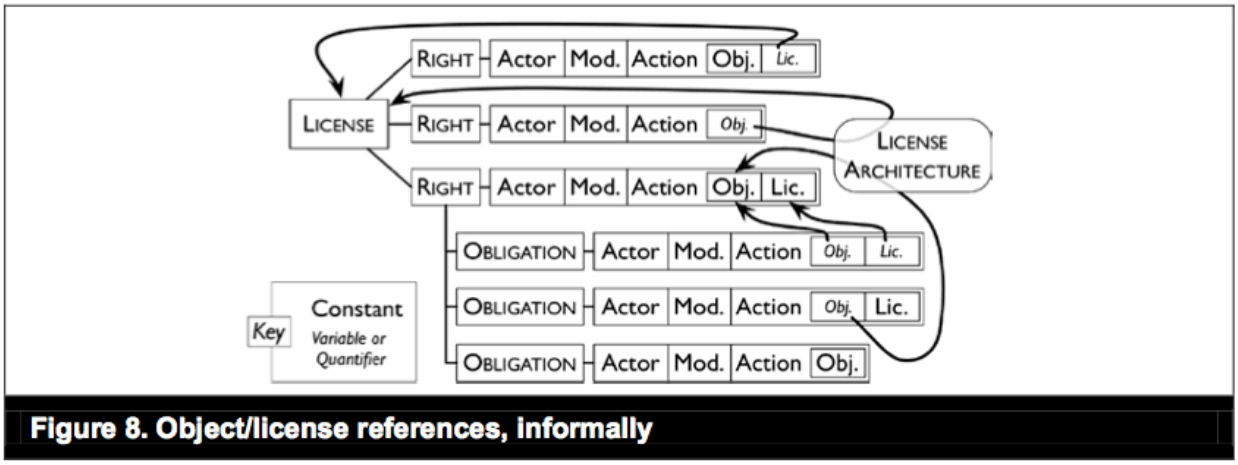


Figure 8. Object/license references, informally

This model of licenses gives a basis for reasoning about licenses in the context of actual systems. The additional information we need about the system is defined by the list of quantifiers that can appear as objects in the rights and obligations. We term this information the *license architecture* (LA). It is an abstraction of the system architecture:

- the set of components of the system;

- the relation mapping each component to its license;
- the relation mapping each component to its set of sources; and
- the relation from each component to the set of components in the same license scope, for each license for which “scope” is defined (e.g. GPL), and from each source to the set of sources of components in the scope of its component.

We tested the internal validity of the study as described above by implementing the meta-model as a Java package, expressing the licenses in the meta-model using the package, automatically re-generating text for each license, and comparing the regenerated text with the original for semantic equivalence (of course the two texts were syntactically dissimilar). At the time of this writing the meta-model has also been validated by a law student specializing in Intellectual Property, an international legal researcher in the area of comparative systems of Intellectual Property, and a law professor in Intellectual Property. These validations were done first informally through discussions of the meta-model and how the legal concepts in licensing and contracts and the specifics of various licenses align with its structures, and then formally through a qualitative analysis in which the legal researcher independently represented two licenses in terms of the meta-model's structure. The key point shown by these validations is that the meta-model is sufficiently expressive to support various legal interpretations. We tested the study's external validity by incorporating the Java package embodying the meta-model into a software architecture tool as described below and by using it to analyze HLS architectures.

In this study we examined ten software licenses; we are continuing on to additional licenses and fully expect that new license structures will be revealed and new elaborations of the meta-model will arise. However, this does not invalidate the results of the present study. We note that any successful theory will have to account for the features we identify here, and at most will add additional features, not remove any.

2.7. License Architecture of Heterogeneously-Licensed Systems

As noted in the previous section, certain elements of software architectures affect how license provisions take effect. A software architecture is composed of *components*, each of which is a "locus of computation and state" in a system, and *connectors* which link them and mediate interactions between them (Shaw et al. 1995). The components and connectors below are significant for one or more FOSS licenses and can affect the license interactions and overall rights

and obligations for the system.

2.7.1. Components

Software source code components. A source code component is one whose source code is available so the component can be modified and rebuilt. These can be either

- standalone programs,
- libraries, frameworks, or middleware,
- inter-application script code that coordinates the actions of two or more applications, or
- intra-application script code that controls actions within its application, as for creating Rich Internet Applications using domain-specific languages such as XUL for the Firefox Web browser (Feldt 2007), or “mashups” that combine data, functionality, or presentations from two or more sources to create a new service (Nelson and Churchill 2006)

Executable components. These components are in binary form and the source code may not be available for access, review, modification, or possible redistribution (Rosen 2005). If proprietary, they often cannot be redistributed, and if so will be present in the design- and run-time architectures but not in the distribution-time architecture.

Software services. An appropriate software service such as Google Docs, provided through a client-server connection to a local or remote server, can be used in place of a source code or executable component.

Application programming interfaces (APIs). Availability of externally visible and accessible APIs is the minimum requirement for an “open system'” (Meyers and Oberndorf 2001). APIs are not and cannot be licensed (Rosen 2005) and can be used to limit the propagation of some license obligations.

Configured system or subsystem architectures. These are software systems that are used as atomic components of a larger system, and whose internal architecture may comprise components with different licenses, affecting the overall system license.

2.7.2. Connectors

Methods of connection. These include

- static linking performed permanently at system build time,
- dynamic linking performed temporarily at run time, and
- client-server connections.

Methods of connection affect license obligation propagation with different methods affecting different licenses. Although there is some controversy over this question, and our statements here should not be taken as legal advice, there appear to be grounds to believe that a client-server connection serves as a license firewall for reciprocal obligations from all current licenses except the Affero GPL (AGPL) version 3.0, which was designed to address client-server connections explicitly, and certain related licenses not otherwise discussed in this article; and more controversially that a dynamic link rather than a static link blocks the same obligations (Determann 2006, Rosen 2007, Stolz 2005). At this writing, no court has ruled definitely on these questions.

Software connectors. Some connectors are themselves software whose intended purpose is to provide a standard or reusable way of communication through common interfaces, e.g. CORBA, Microsoft .NET, Mono, and Enterprise Java Beans. Software connectors, by sending communication through a standard and thus uncopyrightable interface, may affect the propagation of license obligations.

To minimize license interaction, a problematic-licensed component may be surrounded or encapsulated by what we term a *license firewall,* namely a layer of dynamic links, client-server connections, license shims, or other connectors that block the propagation of specific reciprocal obligations.

2.7.3. Other Information in a License Architecture

As noted in the previous section, some license obligations require more information in order to be instantiated, namely the ones that incorporate any object other than *"Right's Object".* A system's license architecture thus includes the relationship from component to sources, which we will not discuss here because it is straightforward from a development point of view. The other such objects, *"X Scope Sources"* and *"X Scope Components*," may be inferred from the information already in the license architecture. We note that we expect the license architecture abstraction to be elaborated in parallel with the meta-model as new license structures are identified.

2.8. Embodying and Applying the Models and Analysis

Figure 9 presents a screenshot of a simple but not trivial HLS that raises the kinds of issues that come up in all HLSs. Its components include a web browser, an email and calendar system, a word processor, and a web application. In the screenshot these four components have been outlined with dashed rectangles. The web browser is Firefox, the email and calendar system is Gnome Evolution, the word processor is AbiWord, and the web application is running in a Red Hat Fedora Linux command window, but similar functionality (with possibly higher quality or better adapted functionality) could be achieved with other components of the same class. This particular system implements a simple office productivity environment for a university research laboratory.

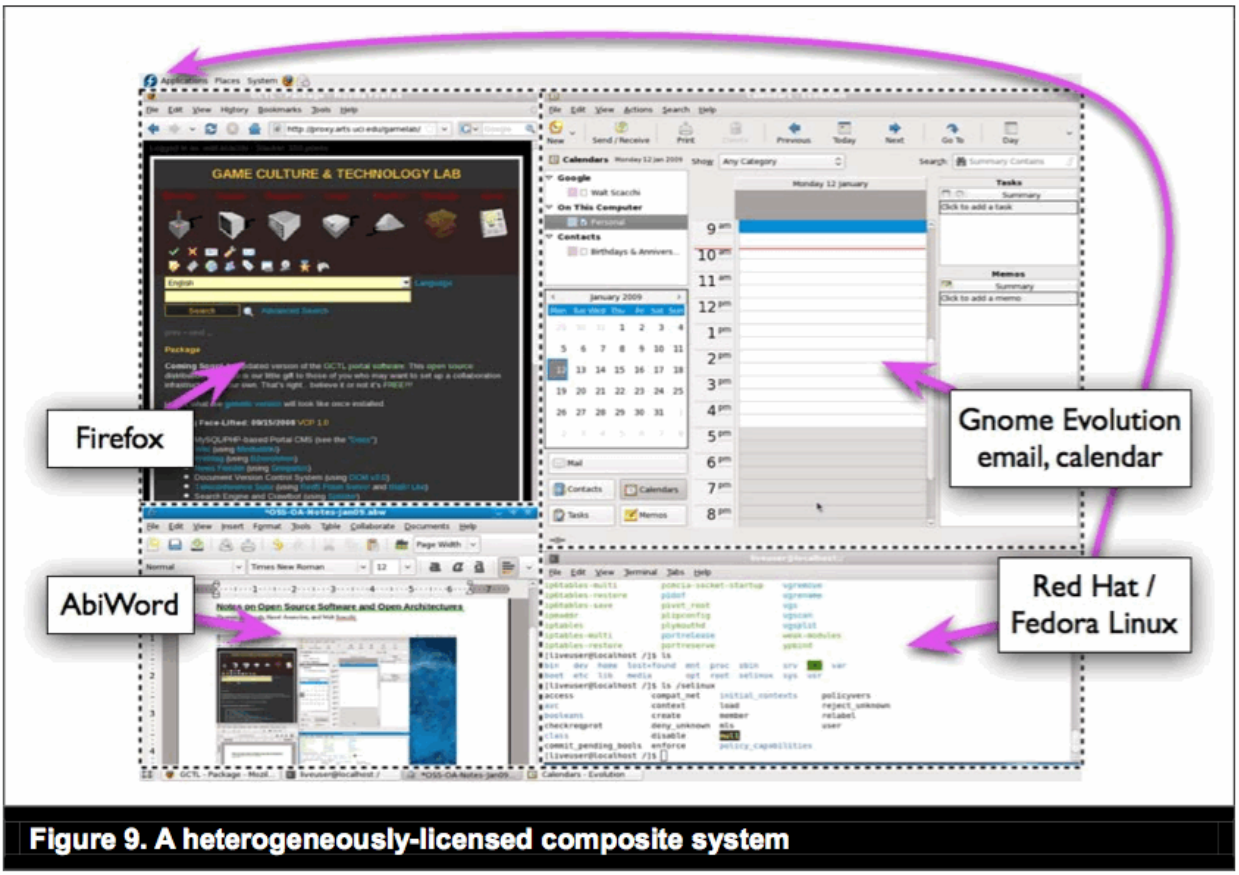


Figure 9. A heterogeneously-licensed composite system

The software architecture of the high level components and their interconnections in this system is shown in Figure 10. Components, shown in upright font, implement pieces of the system's functionality. Each component is associated with a license governing the terms of its use. Connectors, shown in italics, connect and translate if necessary between the interfaces of two components. Where the two interfaces are the same, the connector is identified simply by the API

for the two interfaces and has no license. If the two interfaces differ, then the connector is implemented by a small piece of software, a *shim* that translates between them, and the shim has or could have a license. Arrows show communication between connectors and components. The components, the connectors, and the topology in which they are arranged constitute the architecture of the system. This architecture shows the system's high-level design.

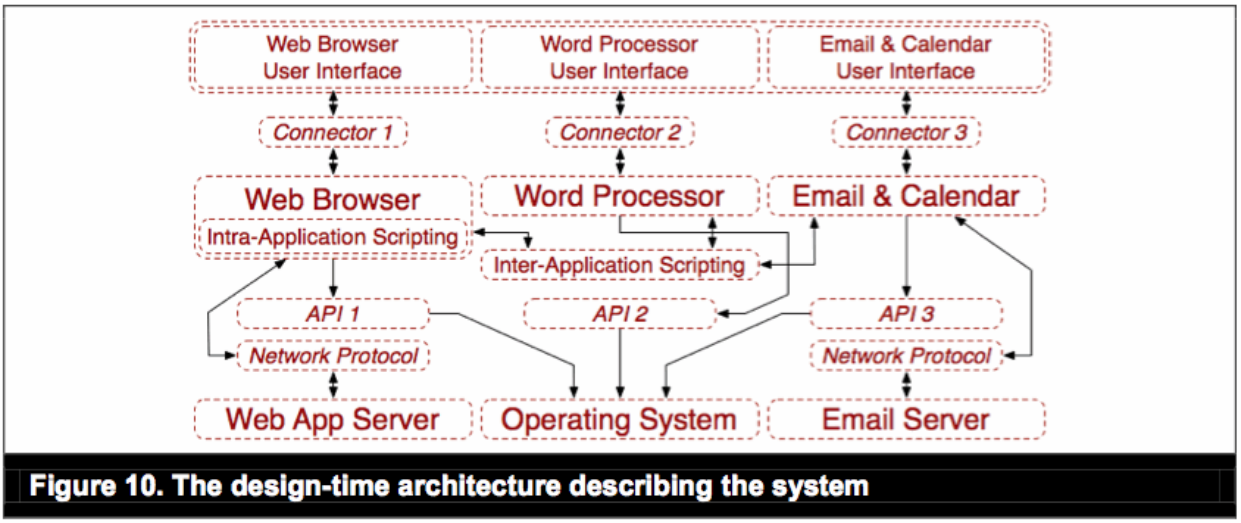


Figure 10. The design-time architecture describing the system

Figure 11 shows the architecture with the abstract components and connectors replaced by specific implementations. In this *instance architecture* describing a particular implementation of the system, the web browser is Firefox, the word processor is AbiWord, connectors 1 through 3 are XWindows calls, and so forth. All of the components and connectors are FOSS, though not homogeneously licensed: for example, the Gnome Evolution and AbiWord components are distributed under GPL version 2.0, whereas the Apache web server component is distributed under the Apache License version 2.0, with which GPL 2.0 is not compatible. The network protocol HTTP, being a data transfer interface conforming to an open standard, has no license. Most of the components and connectors in the architecture are not visible to the user when the system is running, but four of them can be identified in Figure 9, and the user interface (UI) as a whole can be identified as running on a Red Hat Fedora system by the blue script "*f* " in the upper left corner of the screen.

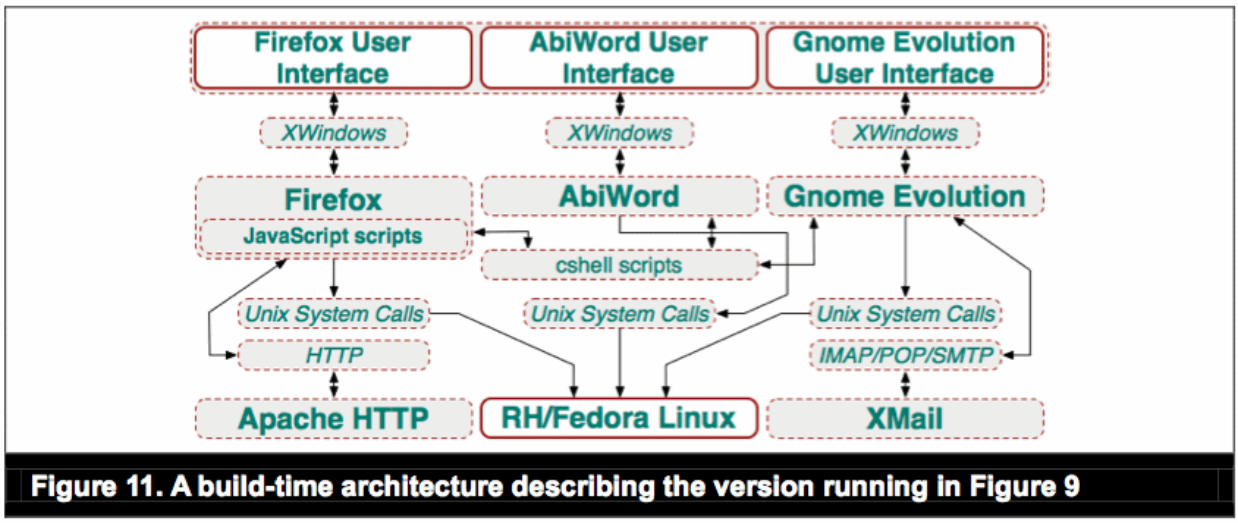


Figure 11. A build-time architecture describing the version running in Figure 9

Figure 12 shows a high-level view of the license architecture of the system of Figure 9, with the mappings to sources and license scopes omitted for clarity. Here each component is represented by its license, with the connectors and connections the same as in the design-time and build-time architectures. For ease of reference, we also provide Figure 13 in which the license architecture is additionally annotated with each component's name.

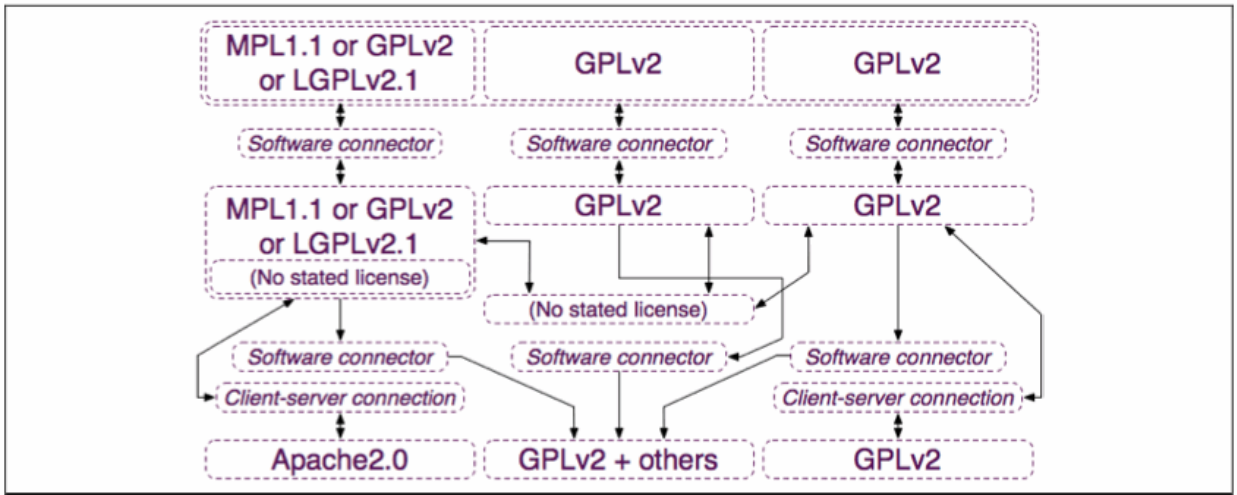


Figure 12. The license architecture of the instantiation of Figure 11

We noted above that GPLv2 and Apache2.0 are not compatible. A reverse-engineering and recovery approach for license analysis, as discussed above in “Related Work” (Gobeille 2008, Tuunanen et al. 2009, Di Penta et al. 2010), could determine that the system in Figures 9-13 *might* have a license conflict but without the license architecture it could not determine whether the system *will* have a conflict. In fact, our analysis (confirmed by the software tool) correctly shows

that no conflict exists because the Apache2.0 component is licensed-firewalled from GPLv2 obligations by the client-server connection that blocks its only path to GPLv2 components. Even without this firewall, the conflict could be avoided by accepting Firefox under MPL1.1, because Firefox is distributed through the Mozilla *tri-license* under which recipients can choose any of the three possible licenses.

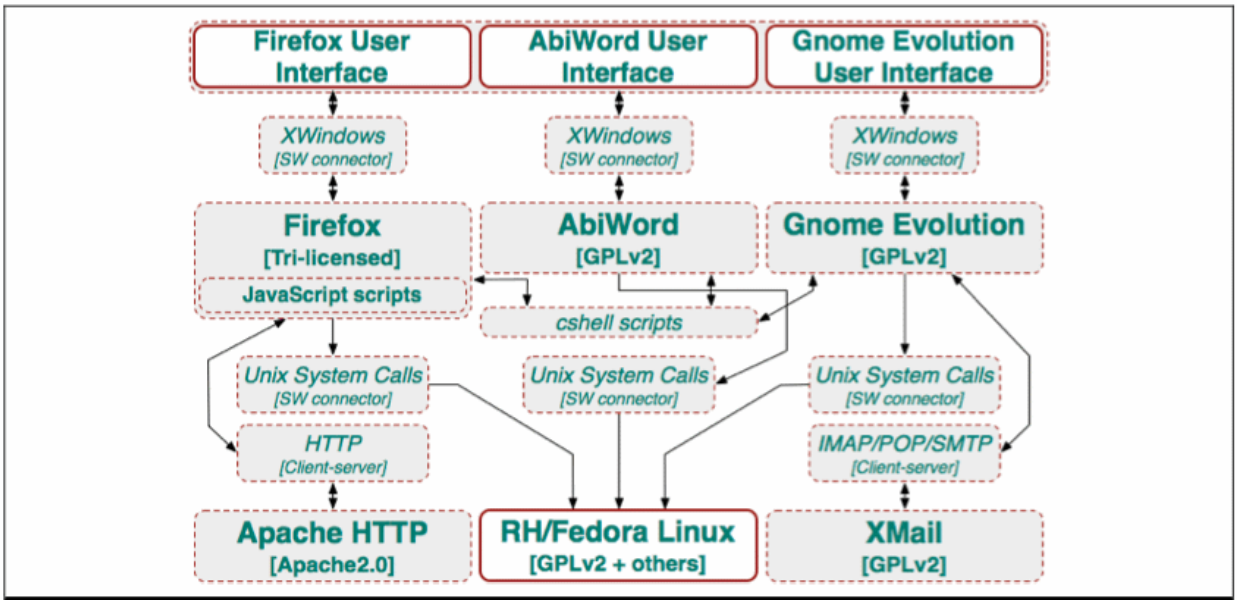


Figure 13. The license architecture, annotated with each component and connector

Because this system has an open architecture, it could be straightforwardly modified to use a proprietary word processor like Corel WordPerfect (or Microsoft Word) in place of AbiWord, for example, to obtain different functionality or to be consistent with an enterprise policy for uniformity across systems. There would be no license conflict because the word processor component in this architecture is firewalled off by client-server connections and software connectors. However, our analysis (confirmed by the tool) identifies that the system no longer has the same rights: anyone can use the system, but the right to distribute the system is no longer available because the Corel Transactional License (CTL) forbids copying and redistribution. This is not a license conflict, but simply the lack of a potentially desired right. Note that the enterprise could choose to purchase a separate copy of WordPerfect for each system, in which case each individual instance of the system could be distributed (although still not copied); or it could distribute the system minus WordPerfect, with each recipient obtaining his/her own copy of WordPerfect and integrating it into the system. We do not maintain that any of these approaches (AbiWord, separate copies of WordPerfect, or user-integrated WordPerfect) is better or worse than another, merely that each has its own distinct functionality, available rights, and/or build process.

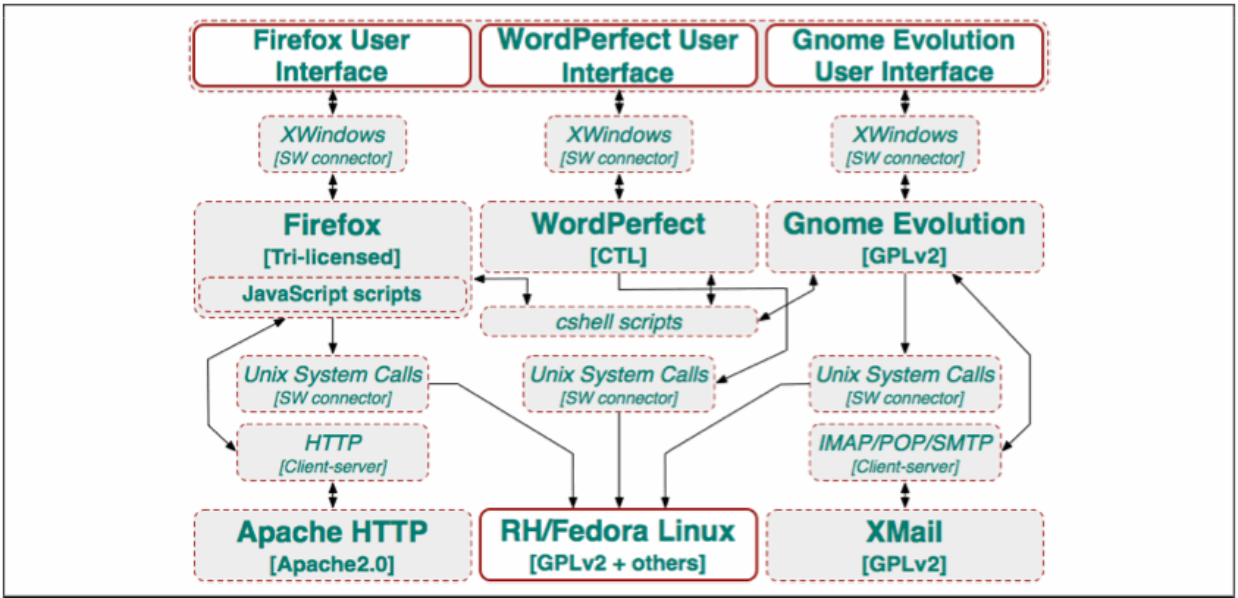


Figure 14. License architecture (annotated) using WordPerfect in place of AbiWord

Finally, we outline the effect of a very different choice: replacing the word processor component with a Web-based word processing service such as Google Docs. In this case the calculation is not dealing only with licenses but also with a Terms of Service agreement whose provisions are different. Some rights are unchanged: anyone can use the system, and the system can be redistributed. But in order to achieve this, not only does the enterprise have to replace the word processing component and the inter-application scripts that shim its interfaces, it has to alter the system's architecture. Google Docs will provide word processing services but only through the web browser, the web browser will not be connected through system calls to the Linux component or through XWindows to the display, and the scripts will be changed to connect to the word processor and browser through the same connection. Again, this choice is not necessarily better or worse than any of the previous three, it merely offers different costs and affordances.

We chose this simple but not trivial example so that the explanation would be of manageable length. Most enterprise HLSs differ from this one only in scope, not in the kinds of issues that arise. The issues would simply be more numerous, and the paths of license effects would be more complex, necessitating a theory-based automated tool not only for ease and accuracy but also to make manageable the balancing and selection among design choices. As we explored and analyzed OA systems, we realized this same example architecture describes a substantial number of e-business information systems. For example, our department's payroll system also has the architecture of Figure 10, although for that system of course the connectors and scripts are

different in order to produce a different specific system function. Using and distributing this or any other HLS within or outside an enterprise has legal implications due to the varied and possibly conflicting licenses found for the systems' components and the large number of distinct obligations that arise. Organizations with assets need to be concerned that the use or distribution of such HLSs can leave them vulnerable to liabilities due to violations (even if inadvertent) of license terms and conditions. Our work gives a basis from which organizations using and developing HLSs can assess their rights and obligations and guide HLS acquisition and development in order to achieve not only the desired system functionality, quality, and evolvability, but also the rights that are needed in exchange for obligations that are acceptable.

2.9. Discussion

Throughout this paper, we have sought to understand and analyze composite software or information systems that constitute an HLS. Here, we discuss some of the consequences that follow from this understanding and analysis.

First, as we saw in related research, there has been a great deal of interest in determining what aspects or features of FOSS development projects are significant contributors to project success. The choices among available FOSS licenses, when applied to a single system as a whole, revealed that certain license choices are positively associated with project success. But, what are we to expect in trying to anticipate potential outcomes when system components may have different licenses? This is part of the challenge for understanding HLSs. Consequently, it is unclear what outcomes for HLS development projects should be anticipated based on prior studies of FOSS success factors, and thus this should be recognized as an outstanding problem for further study. However, it does appear that there is growing awareness that both public and private sector enterprises are moving to develop composed (or otherwise integrated) systems of systems and that such HLSs may become more commonplace.

Second, through modeling of FOSS licenses and analysis of HLSs rendered as an OA with a license attributed to each component and to the other components to which it is directly linked (or interconnected), it becomes possible to articulate which licenses or constraints match or conflict with one another. The case study in the previous section helps demonstrate this. Further, when licenses' constraints are discovered, the nature of each conflict (e.g., "must not redistribute software" versus "must redistribute software if modified") then gives rise to possible ways in which

that conflict can be mitigated or resolved. These include reconfiguring the system architecture, replacing conflicting components with compatible alternatives under non-conflicting licenses, or inserting license mediators. However, if an OA for the system is not available and cannot be derived from available information (as is the case for the Unity3D software package identified in the Introduction), then it may not be possible to analyze or reliably determine what license constraints apply, or whether any are in conflict, and this places system consumers or users in an uncertain situation. Thus, understanding and analyzing HLSs requires a model of the constraints found in the system component licenses, along with an OA rendering of the system and corresponding license constraints.

Lastly, once the analysis of an HLS has been conducted and has identified license conflicts resolved or eliminated, then what remains is a set of license constraints that characterize the rights provided and the obligations to be fulfilled by the system's consumers or users. But this unified set of license constraints in general does not conform to an existing single license (because if it did, the system would not be considered an HLS), so then what is the license in effect? We believe one way to address this is to consider the resulting set of non-conflicting rights and obligations for an HLS to constitute a new kind of license which we can designate as a virtual license. However, we do not anticipate that such a virtual license will simply be treated as a new type of license (and thus submitted to a license review authority like the Open Source Initiative). Instead, a virtual license will simply represent a set of rights and obligations within corporate policy space or license regime that an enterprise finds acceptable, manageable, and traceable (cf. Dinkelacker et al. 2004). In this regard, we may expect that future study could determine whether enterprises can simply specify which resulting license rights or obligations they will accept and fulfill, and which they will not, independent of the name or pedigree of the FOSS license or project from which those rights and obligations may have originated.

2.10. Conclusion

In this paper, we introduced the problem of how to understand and analyze heterogeneously-licensed systems composed and built from FOSS and/or proprietary components, each of which may be subject to different licenses with different rights and obligations. We started by providing examples of HLSs that are beginning to appear, and that help articulate the challenge of understanding how license constraints may interact, match, or conflict within the system's architecture, whether at design-time, build-time, or run-time. We found that such analysis requires

an explicit semantic model of the license rights and obligations along with an open architectural rendering that can be attributed with the corresponding license constraints. Without both of these, it may not be possible to systematically analyze or comprehensively understand what license constraints apply to the whole system (or system of systems). But with both, we demonstrated that it is possible to conduct such an analysis and to determine which license constraints match, subsume, or conflict with one another. It also becomes possible to explore and readily evaluate alternative system architectures that may resolve conflicting license constraints, particularly when analysis of HLSs can be supported with an automated environment, such as the one we have developed to this end.

The contribution presented in this paper constitutes identification of a new, emerging category of problem, which we identify as the challenge of heterogeneously-licensed systems, along with an empirically grounded approach and design theory for analyzing and modeling FOSS license rights and obligations, and to modeling and analyzing the architecture of an HLS, in order to determine the resulting set of license constraints that applies to the composed system. We described and demonstrated our approach, along with the kinds of information we employ, which are empirically observable when FOSS components are employed and when all participating software license rights and obligations can be systematically modeled. Whether our approach is relevant to legal interpretations or framings for the purpose of asserting defensible intellectual property rights is beyond the scope of our study (and expertise) and thus remains an open question for future study.

THIS PAGE INTENTIONALLY LEFT BLANK

Acknowledgments

The authors extend their thanks to the anonymous reviewers of earlier versions of this paper for their thoughtful and insightful evaluations.

Support for this research is through grant #0808783 from the National Science Foundation, and also #N00244-10-1-0038 from the Acquisition Research Program at the Naval Postgraduate School. No review, approval, or endorsement implied.

THIS PAGE INTENTIONALLY LEFT BLANK

2.11. References

Adobe (2009). Interapplication Communication (IAC). http://www.adobe.com/devnet/acrobat/interapplication.html.

Allen, L. E. and Saxon, C. S. (1995). Better language, better thought, better communication: the A-Hohfeld language for legal analysis. In *5th International Conference on Artificial Intelligence and Law (ICAIL'95),* pages 219–228.

American Law Institute (1981). *The Restatement (Second) of Contracts*.

Alspaugh, T. A., Asuncion, H.U., and Scacchi, W. (2009a). Intellectual Property Rights Requirements for Heterogeneously-Licensed Systems. In *17th IEEE International Requirements Engineering Conference (RE'09),* pages 24–33. (2009a).

Alspaugh, T. A., Asuncion, H. U., and Scacchi, W. (2009b). The Role of Software Licenses in Open Architecture Ecosystems. In *First International Workshop on Software Ecosystems (IWSECO-2009),* pages 4–18.

Apache Software Foundation (2009). Individual Contributor License Agreement version 2.0. http://www.apache.org/licenses/icla.txt

Asay, M. (2008). In Acquiring Merrill Lynch, must Bank of America open source its software? CNET News, http://news.cnet.com/8301-13505_3-10043029-16.html, 16 September 2008.

Balkin, J. M. (1991). The promise of legal semiotics. University *of Texas Law Review,* 69:1831–1852.

Basili,V. R., Caldiera, G., and Rombach, H. D. (1994). The Goal Question Metric approach. In *Encyclopedia of Software Engineering*, pages 528–532, John Wiley and Sons.

Black Duck Software (2009). Top 20 Most Commonly Used Licenses in Open Source Projects. http://www.blackducksoftware.com/oss/licenses

Breaux, T. D., Antón, A. I., and Doyle, J. (2008). Semantic parameterization: A process for modeling domain descriptions. *ACM Transactions on Software Engineering and Methodology,* 18(2).

Brown, A.W., and Booch, G. (2002). Reusing open-source software and practices: The impact of open-source on commercial vendors. In: *Software Reuse: Methods, Techniques, and Tools (ICSR-7)*.

Corbin, J. M. and Strauss, A. C. (2007). *Basics of Qualitative* Research*: Techniques and procedures for Developing Grounded Theory.* SAGE Publications.

Corel Transactional License (2009). http://apps. corel.com/clp/terms.html.

Creswell, J. W. (2003). *Research Design: Qualitative, Quantitative, and Mixed Methods*

Approaches. SAGE Publications, Thousand Oaks, CA, USA.

Daskalopulu, A. and Sergot, M. (1997). The Representation of Legal Contracts. *AI & Society,* 11(1–2):6-17.

de Laat, P.B. (2007). Governance of open source software: state of the art, *J. Management and Governance,* 11(2), 165–177.

Determann, L. (2006). Dangerous Liaisons—Software Combinations as Derivative Works? Distribution, Installation, and Execution of Linked Programs Under Copyright Law, Commercial Licenses, and the GPL. *Berkeley Technology Law Journal,* 21(4).

Di Penta, M., German, D. M., Gueheneuc, Y.-G., and Antoniol, G. (2010). An Exploratory Study of the Evolution of Software Licensing. *Proc. 29th International Conference on Software Engineering (ICSE '10).*

Dinkelacker, J., Garg, P. K., Miller, R., and Nelson, D. (2002). Progressive Open Source, *Proc. 24th Intern. Conf. Software Engineering*, Orlando, FL, 177–184.

Elliott, M. S., and Scacchi, W. (2003). Free software developers as an occupational community: resolving conflicts and fostering collaboration. *ACM International Conference on Supporting Group Work (GROUP'03)*, pages 21–30.

Elliott, M. S., and Scacchi, W. (2008). Mobilization of software developers: the free software movement. *Information Technology & People*, 21(1):4–33.

Feldt, K. (2007). *Programming Firefox: Building Rich Internet Applications with XUL.* O'Reilly Media.

Fontana, R., Kuhn, B.M., Molgen, E. et al. (2008). *A Legal Issues Primer for Open Source and Free Software Projects*, Software Freedom Law Center, Version 1.5.1. http://www.softwarefreedom.org/resources/2008/foss-primer.pdf

Free Software Foundation (1991). GNU General Public License Version 2. http://opensource.org/licenses/gpl-2.0.php

Free Software Foundation (1999). GNU Lesser General Public License Version 2.1. http://opensource.org/licenses/lgpl-2.1.php

Free Software Foundation (2007a). GNU Affero General Public License Version 3. http://opensource.org/licenses/agpl-v3.html

Free Software Foundation (2007b). GNU General Public License Version 3. http://opensource.org/licenses/gpl-3.0.html

Free Software Foundation (2007c). GNU Lesser General Public License Version 3. http://opensource.org/licenses/lgpl-3.0.html

German, D. M. and Hassan, A. E. (2009). License integration patterns: Addressing license

mismatches in component-based development. In *Proc. 31st International Conference on Software Engineering (ICSE '09),* Vancouver, BC, Canada, 188–198.

Glaser, B. G. and Strauss, A. L. (1967). *The* Discovery *of Grounded Theory: Strategies for Qualitative Research.* Aldine Publishing Company.

Gobeille, R. (2008). The FOSSology project. In *International Working Conference on Mining Software Repositories (MSR'08),* pages 47–50.

Gordon, W. J. (1989). An Inquiry into the Merits of Copyright: The Challenges of Consistency, Consent, and Encouragement Theory. *Stanford Law Review,* 41(6):1343–1469.

Guadamuz, A. (2009). The License/Contract Dichotomy in Open Licenses: A Comparative Analysis. *University of La Verne Law Review,* 30(2):101–116.

Hevner, A. R., March, S. T., Park, J., and Ram, S. (2004). Design Science in Information Systems Research. *MIS Quarterly,* 28(1):75–105.

Hillman, R. A. and O'Rourke, M. A. (2009). Rethinking Consideration in the Electronic Age. *Hastings Law Journal,* 61:311–336.

Hohfeld, W. N. (1913). Some Fundamental Legal Conceptions as Applied in Judicial Reasoning. *Yale Law Journal,* 23(1):16–59.

Huhns, M. N. and Singh, M. P. (1998). Agent Jurisprudence. *IEEE Internet Computing,* 2(2):90–91.

Humphris-Norman, D. O.. *Justice, rights, and jural relations: a philosophy of justice and its relationships.* University of Southampton. 2009.

Kemp, R. (2009). Current developments in Open Source Software. *Computer Law and Security Review,* 25(6):569–582.

McCarty, L. T. (2002). Ownership: A case study in the representation of legal concepts. *Artificial Intelligence and Law,* 10(1-3):135–161.

Meyers, B. C. and Oberndorf, P. (2001). *Managing Software Acquisition: Open Systems and COTS Products.* Addison-Wesley Professional.

Miles, M. B. and Michael Huberman, M. (1994). *Qualitative Data Analysis: An expanded sourcebook.* SAGE Publications.

Mozilla Foundation (2009). Mozilla Code Licensing. http://www.mozilla.org/MPL/.

MySQL (2006). MySQL Licensing Policy. http://www.mysql.com/about/legal/licensing/

Nelson, L. and Churchill, E. F. (2006). Repurposing: Techniques for reuse and integration of interactive systems. In *International Conference on Information Reuse and Integration (IRI-08),* page 490.

OSI (2009). Open Source Initiative. http://www.opensource.org/.

Oreizy, P. (2000). *Open Architecture Software: A Flexible Approach to Decentralized Software*

Evolution. PhD Thesis, Information and Computer Science, University of California. http://www.ics.uci.edu/~peymano/papers/thesis.pdf

Roberts, J. A., Hann, I.-H., and Slaughter, S. A. (2006). Understanding the Motivations, Participation, and Performance of Open Source Software Developers: A Longitudinal Study of the Apache Projects. *Management Science,* 52(7):984–999.

Rosen, L. (2007). Comments on GPLv3. http://www.rosenlaw.com/GPLv3-Comments.htm

Rosen, L. (2005). *Open Source Licensing: Software Freedom and Intellectual Property Law.* Prentice Hall.

Scacchi, W. and Alspaugh, T.A. (2008). Emerging Issues in the Acquisition of Open Source S Software within the U.S. Department of Defense, *Proc. 5th Annual Acquisition Research Symposium*, Vol. 1, 230–244, Naval Postgraduate School, Monterey, CA.

Sen, R. (2007). A Strategic Analysis of Competition Between Open Source and Proprietary Software, *J. Management Information Systems*, 24(1), 233–257. Summer.

Sen, R., Subramaniam, C., and Nelson, M. (2008). Determinants of the Choice of Open Source Software Licenses, *J. Management Information Systems*, 25(3), 207–240, Winter.

Shaw, M., DeLine, R., Klein, D. V., Ross, T. L., Young, D. M., and Zelesnik, G. (1995). Abstractions for Software Architecture and Tools to Support Them. *IEEE Transactions on Software Engineering,* 21(4):314–335.

Siena, A., Mylopoulos, J., Perini, A., and Susi, A. (2008). From Laws to Requirements. In *First International Workshop on Requirements Engineering and Law (RELAW'08),* 6–10.

Stewart, K.J., Ammeter, A.P., and Maruping, L.M. (2006). Impacts of License Choice and Organizational Sponsorship on User Interest and Development Activity in Open Source Software Projects, *Information Systems Research*, 17(2), 126–144, June.

Stoltz, M. L. (2005). The Penguin Paradox: How the Scope of Derivative Works in Copyright Affects the Effectiveness of the GNU GPL. *Boston University Law Review,* 85(5):1439–1477.

Subramaniam, C., Sen. R., and Nelson, M. (2009). Determinants of Open Source Software Project Success: A Longitudinal Study, *Decision Support Sys.*, 46(2), 576–585.

St. Laurent, A. M. (2004). *Understanding Open Source and Free Software Licensing.* O'Reilly Media, Inc., Sebastopol, CA.

Tuunanen, T., Koskinen, J., and Kärkkäinen T. (2009). Automated software license analysis. *Automated Software Engineering*, 16(3-4):455–490.

Unity Technologies (2009). Unity End User License Agreement.

http://unity3d.com/unity/ unity-end-user-license-2.x.html.

U.S. Copyright Act (2009). 17 U.S.C. http://www.copyright.gov/title17/. University of California, Berkeley (1998). The BSD License. http://opensource.org/licenses/bsd-license.php

Ven, K. and Mannaert, H. (2008). Challenges and strategies in the use of open source software by independent software vendors. *Information and Software Technology,* 50(9-10), 991–1002.

THIS PAGE INTENTIONALLY LEFT BLANK

3. Understanding the Role of Licenses and Evolution in Open Architecture Software Ecosystems

Walt Scacchi
Institute for Software Research. University of California, Irvine USA

Thomas A. Alspaugh
Computer Science Dept., Georgetown University, Washington, DC, USA

November 2010

Abstract

The role of software ecosystems in the development and evolution of heterogeneously-licensed open architecture systems has received insufficient consideration. Such systems are composed of components potentially under two or more licenses, open source or proprietary or both, in an architecture in which evolution can occur by evolving existing components, replacing them, or refactoring. The software licenses of the components both facilitate and constrain the system's ecosystem and its evolution, and the licenses' rights and obligations are crucial in producing an acceptable system. Consequently, software component licenses and the architectural composition of a system determine the software ecosystem niche where a system lies. Understanding and describing software ecosystem niches is a key contribution of this work. A case study of an open architecture software system that articulates different niches is employed to this end. We examine how the architecture and software component licenses of a composed system at design-time, build-time, and run-time help determine the system's software ecosystem niche and provide insight and guidance for identifying and selecting potential evolutionary paths of system, architecture, and niches.

Published as: W. Scacchi and T.A. Alspaugh, Understanding the Role of Licenses and Evolution in Open Architecture Software Ecosystems, submitted for publication, (in revision, November 2010).

3.1. Introduction

A substantial number of development organizations are adopting a strategy in which a software-intensive system (one in which software plays a crucial role) is developed with an *open architecture* (OA) [27], whose components may be open source software (OSS) or proprietary with open application programming interfaces (APIs). Such systems evolve not only through the evolution of their individual components, but also through replacement of one component by another, possibly from a different producer or under a different license. With this approach, another organization often comes between software component producers and system consumers. These organizations take on the role of system architect or integrator, either as independent software vendors, government contractors, system integration consultants, or in-house system integrators. In turn, such an integrator designs a system architecture that can be composed of components largely produced elsewhere, interconnected through interfaces accommodating use of dynamic links, intra-application or inter-application scripts, communication protocols, software buses, databases/repositories, plug-ins, libraries, or software shims as necessary to achieve the desired result. An OA development process results in an ecosystem in which the integrator is influenced from one direction by the goals, interfaces, license choices, and release cycles of the software component producers, and from another direction by the needs of the system's consumers. As a result, the software components are reused more widely and the resulting OA systems can achieve reuse benefits such as reduced costs, increased reliability, and potentially increased agility in evolving to meet changing needs. An emerging challenge is to realize the benefits of this approach when the individual components are heterogeneously licensed [4, 15, 31], each potentially with a different license, rather than a single OSS license as in uniformly-licensed OSS projects or a single proprietary license as in proprietary development.

This challenge is inevitably entwined with the software ecosystems that arise for OA systems (Figure 1). We find that an OA software ecosystem involves organizations and individuals producing, composing, and consuming components that articulate software supply networks from producers to consumers, but also:

- the OA of the system(s) in question,
- the open interfaces met by the components,
- the degree of coupling in the evolution of related components, and
- the rights and obligations resulting from the software licenses under which various components are released, that propagate from producers to consumers.

These four items play a key role in determining the software ecosystem for a specific system, as

Figures 2, 3, and 4 below illustrate and the remainder of this paper will make clear.

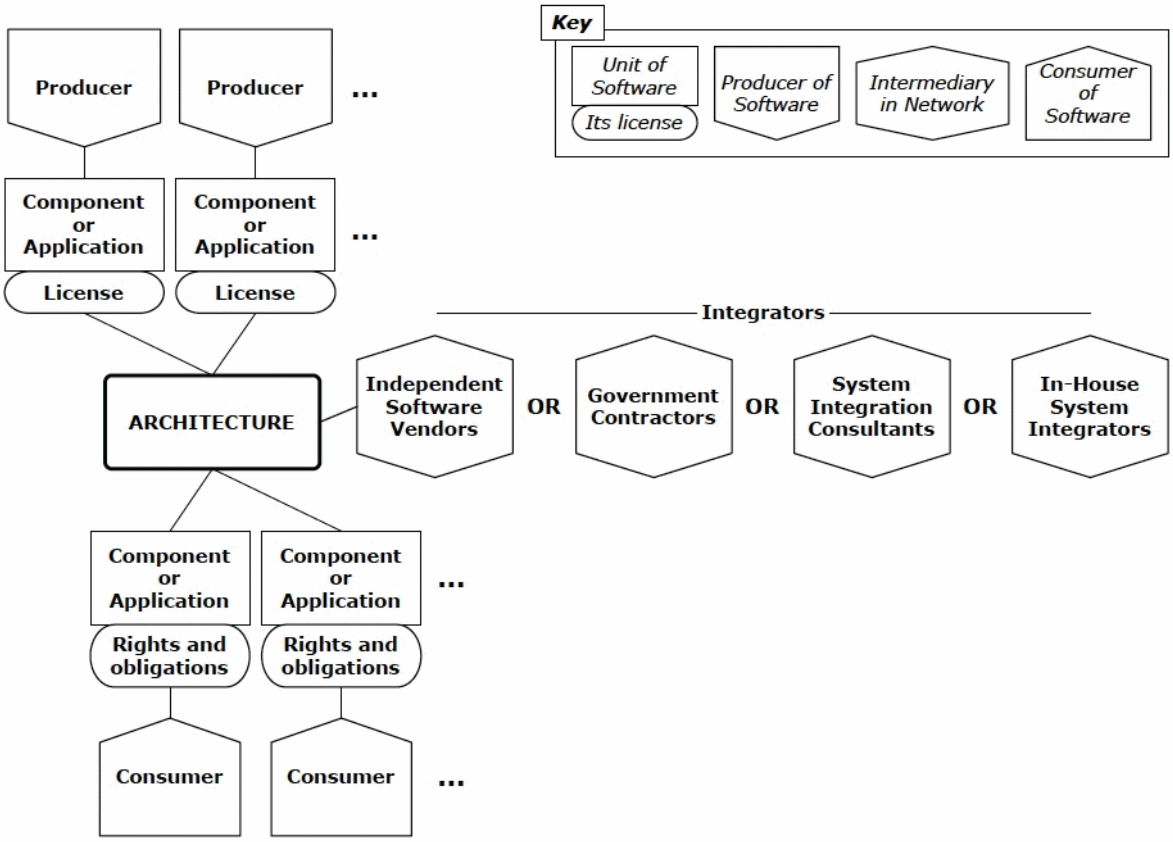


Figure 1: Schema for OA software supply networks (notation follows [10])

3.1.1. A Motivating Example

A motivating example of this approach is the *Unity* game development tool, produced by Unity Technologies [33]. Its license agreement, from which we quote below, lists eleven distinct licenses and indicates that the tool is produced, apparently using an OA approach, using at least 18 externally produced components or groups of components:

1. The Mono Class Library, Copyright 2005-2008 Novell, Inc.
2. The Mono Runtime Libraries, Copyright 2005-2008 Novell, Inc.
3. Boo, Copyright 2003-2008 Rodrigo B. Oliveira
4. UnityScript, Copyright 2005-2008 Rodrigo B. Oliveira
5. OpenAL cross platform audio library, Copyright 1999-2006 by authors.

6. PhysX physics library. Copyright 2003-2008 by Ageia Technologies, Inc.
7. libvorbis. Copyright (c) 2002-2007 Xiph.org Foundation
8. libtheora. Copyright (c) 2002-2007 Xiph.org Foundation
9. zlib general purpose compression library. Copyright (c) 1995-2005 Jean-loup Gailly and Mark Adler
10. libpng PNG reference library
11. jpeglib JPEG library. Copyright (C) 1991-1998, Thomas G. Lane.
12. Twilight Prophecy SDK, a multi-platform development system for virtual reality and multimedia. Copyright 1997-2003 Twilight 3D Finland Oy Ltd
13. dynamic bitset, Copyright Chuck Allison and Jeremy Siek 2001-2002.
14. The Mono C# Compiler and Tools, Copyright 2005-2008 Novell, Inc.
15. libcurl. Copyright (c) 1996-2008, Daniel Stenberg <daniel@haxx.se>.
16. PostgreSQL Database Management System
17. FreeType. Copyright (c) 2007 The FreeType Project (www.freetype.org).
18. NVIDIA Cg. Copyright (c) 2002-2008 NVIDIA Corp.

The software ecosystem for *Unity* as a standalone software package is delimited by the diverse set of software components listed above. However, the architecture that integrates or composes these components is closed and thus unknown to a system consumer as is the manner in which the different licenses associated with these components impose obligations or provide rights to consumers or on the other components to which they are interconnected. Subsequently, we see that software ecosystems can be understood in part by examining relationships between architectural composition of software components that are subject to different licenses, and this necessitates access to the system's architecture composition. By examining the open architecture of a specific composed software system, it becomes possible to explicitly identify the software ecosystem niche in which the system is embedded.

3.1.2. Understanding Open Architecture Software Ecosystems

A software ecosystem constitutes a software supply network that connects software producers to integrators to consumers through licensed components and composed systems. Figure 2 outlines the software ecosystem elements and relationships for an OA case study that we examine in this paper.

A *software ecosystem niche* articulates a specific software supply network that interconnects

particular software producers, integrators, and consumers. A software system defined niche may lie within an existing single ecosystem or one that may span a network of software producer ecosystems. A composed software system architecture helps determine the software ecosystem niche, since the architecture identifies the components, their licenses and producers, and thus the network of software ecosystems in which it participates. Such a niche also transmits license-borne obligations and access/usage rights passed from the participating software component producers through integrators and to system consumers. Thus, system architects or component integrators help determine in which software ecosystem niche a given instance architecture for the system

participates.

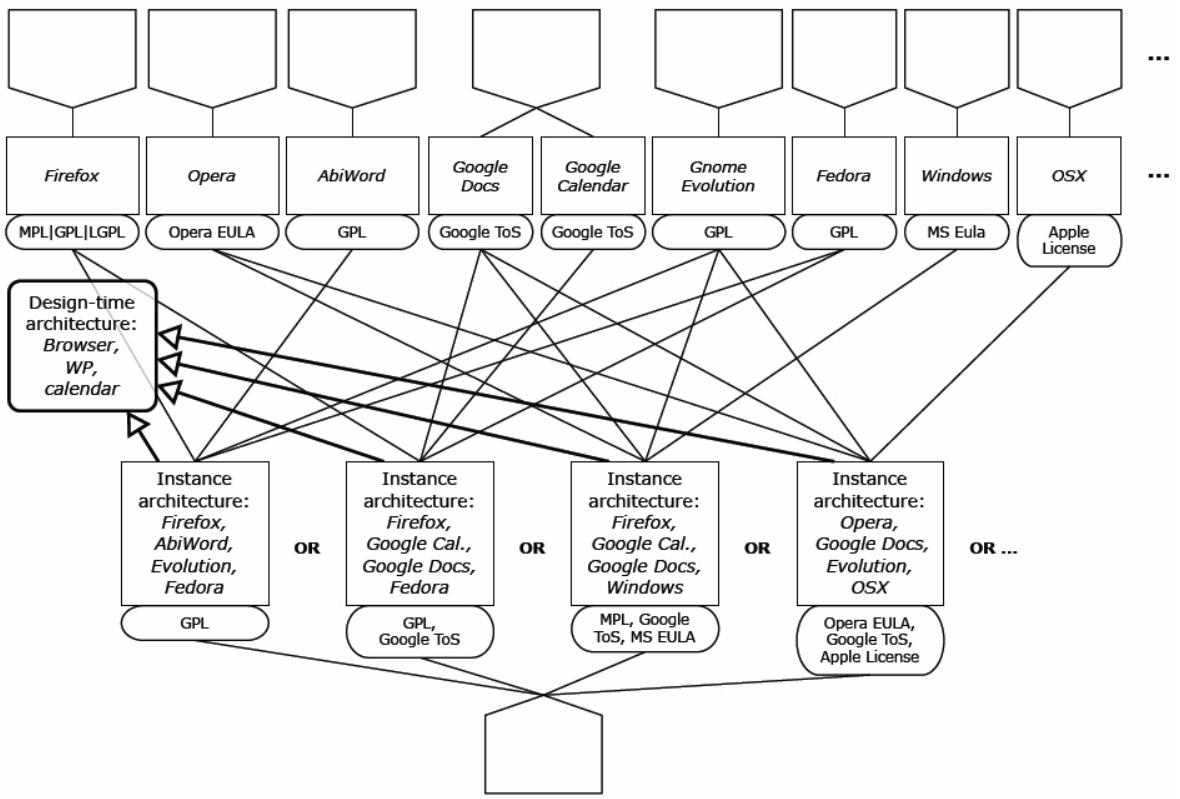


Figure 2: Ecosystem for three possible instantiations of a single design architecture

As a software system evolves over time, as its components are updated or changed, or their architectural interconnections are re-factored, it is desirable to determine whether and how the system's ecosystem niche may have changed. A system may evolve because its consumers want to migrate to alternatives from different component producers or choose components whose licenses are more/less desirable. Software system consumers may want to direct their system

integrators to compose the system's architecture so as to move into or away from certain niches. Consequently, system integrators can update or modify system architectural choices at design-time, build-time (when components are compiled together into an executable), or run-time (when bindings to remote executable services are instantiated) to move a software system from one niche to another. Thus, understanding how software ecosystem niches emerge is a useful concept that links software engineering concerns for software architecture, system integration/composition, and software evolution to organizational and supply network relationships between software component producers, integrators, and system consumers.

To help explain how OA systems articulate software ecosystem niches, we provide a software architecture case study for use in this paper. Its architectural design comprises a web browser, word processor, calendaring and email applications, host platform operating system, and remote services as its components. This system is intentionally simple for expository purposes, but its ecosystem is complex in important ways:

- Alternatives exist for each component that involve diverse possibilities for licenses, evolution paths, system capabilities, requirements, and ecosystems, such as *Word* (proprietary), *AbiWord* (OSS), or *Google Docs* (service) for the word processor.
- Some component choices co-evolve with coordination among suppliers (such as Mozilla and Gnome components) while others do not (Section 5).
- The system in its current open architecture is independent of any one vendor. Such ecosystems are more revealing and offer more evolution paths for study (and use) than a system in an ecosystem dominated by a single vendor such as Microsoft, Oracle, or SAP. Single-vendor-dominated ecosystems may be larger, but are less diverse and thus less interesting and offer fewer choices with significant ecosystem impact.
- The system is independent of any one platform; for example, it could be evolved to run on a mobile device (Section 6), moving it into a much different niche.
- The insights provided by the case study system allow one, we believe, to anticipate or even predict the kinds of issues that will arise when new platforms emerge.

The four primary components collectively represent more than a million lines of code. Each component, and its sub-components recursively down to the smallest, is a composition of other more primitive components that may be independently developed or developed as part of this system, and which may be added to the ecosystem relationships in order to consider its effect on supply chains and evolution. An individual component such as *Firefox* constitutes a micro-platform

itself on which Ajax, Rich Internet Applications, or other scripted functionality (e.g. invoking an embedded link to a *YouTube* Video player) can run internally, constituting an embedded ecosystem. Equivalent components from different OSS or proprietary software producers can be identified, where each alternative is subject to a different type of software license. For example, for Web browsers, we consider the *Firefox* browser from the Mozilla Foundation, which comes with a choice of OSS license (MPL, GPL, or LGPL), and the *Opera* browser from Opera Software, which comes with a proprietary software end-user license agreement (EULA). Similarly, for word processor, we consider the OSS *AbiWord* application (GPL) and Web-based *Google Docs* service (proprietary Terms of Service).

The OA we describe covers a number of systems we have identified, built, and deployed in a university research laboratory. Example niches involving these components appear in Figures 3

and 4.

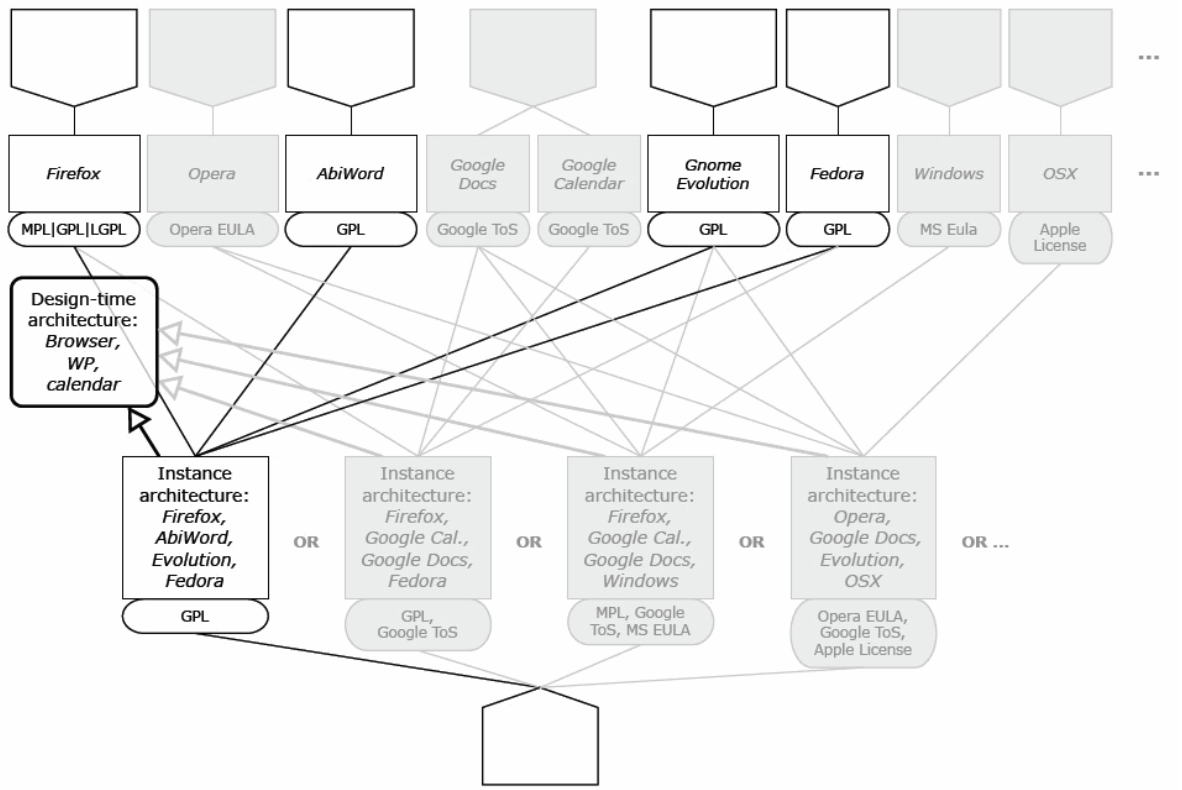


Figure 3: The ecosystem niche for one instance architecture

We have also developed OA systems with more complex architectures that incorporate components for content management systems (*Drupal*), wikis (*MediaWiki*), blogs (*B2evolution*), teleconferencing and media servers (*Flash Media Server*, *Red5 media server*), chat (*BlaB! Lite*),

Web spiders and search engines (*Nutch*, *Lucene*, *Sphider*), relational database management systems (*MySQL*), and others. Furthermore, the OSS application stacks and infrastructure (platform) stacks found at BitNami.org/stacks (accessed 29 April 2010) could also be incorporated in OA systems, as could their proprietary counterparts. However, these more complex OAs still reflect the core architectural concepts and constructs, as well as the software ecosystem relationships, that we present in our example case study in a more accessible manner.

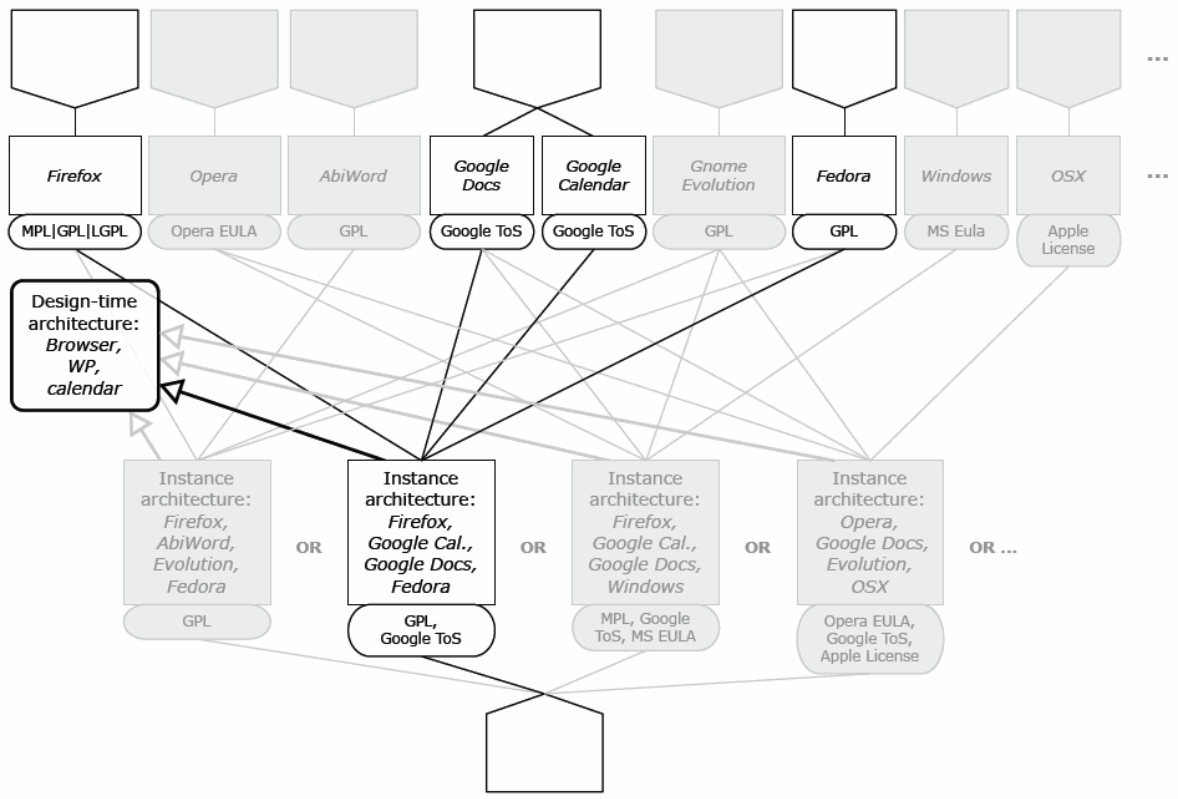


Figure 4: The ecosystem niche for a second instance architecture

The software ecosystem niches for the case study system, or indeed any system, depend on which component implementations are used and the architecture in which they are combined and instantiated, as does the overall rights and obligations for the instantiated system. In addition, we build on previous work on heterogeneously-licensed systems [4, 15, 31] by examining how OA development affects and is affected by software ecosystems and the role of component licenses in shaping OA software ecosystem niches.

Consequently, we focus our attention to understand the ecosystem of an open architecture

software system such that:

- It must rest on a license structure of rights and obligations (Section 4), focusing on obligations that are enactable and testable.[4]
- It must take account of the distinctions between the design-time, build-time, and distribution-time architectures (Section 3) and the rights and obligations that come into play for each of them.
- It must distinguish the architectural constructs significant for software licenses, and embody their effects on rights and obligations (Section 3).
- It must define license architectures.
- It must account for alternative ways in which software systems, components, and licenses can evolve (Section 5), and
- It must provide an automated environment for creating and managing license architectures. We are developing a prototype that manages a license architecture as a view of the system architecture [2].

The remainder of this paper is organized as follows. Section 2 places this work in the context of related research. Section 3 discusses open architecture, and the influence of software licenses is discussed in Section 4. Section 5 addresses evolution of software ecosystems. Section 6 discusses some implications that follow from this study, and Section 7 concludes the paper.

3.2. Related Research

The study of software ecosystems is emerging as an exciting new area of systematic investigation and conceptual development within software engineering. Understanding the many possible roles that software ecosystems can play in shaping software engineering practice is gaining more attention since the concept first appeared [21]. Bosch [8] builds a conceptual lineage from software product line (SPL) concepts and practices [7, 12] to software ecosystems. SPLs focus on the centralized development of families of related systems from reusable components hosted on a common platform with an intra-organizational base, with the resulting systems either intended for in-house use or commercial deployments. Software ecosystems then are seen to extend this practice to systems hosted on an inter-organizational base, which may resemble development

4 For example, many OSS licenses include an obligation to make a component's modified code public, and whether a specific version of the code is public at a specified Web address is both enactable (it can be put into practice) and testable. In contrast, the General Public License (GPL) v.3 provision "No covered work shall be deemed part of an elective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty" is not enactable in any obvious way, nor is it testable — how can one verify what others deem?

approaches conceived for virtual enterprises for software development [24]. Producers of commercial software applications or packages thus need to adapt their development strategy and business model to one focused on coordinating and guiding decentralized software development of its products and enhancements (e.g., plug-in components).

Along with other colleagues [9, 11, 34], Bosch identifies alternative ways to connect reusable software components through integration and tight coupling found in SPLs, or via loose coupling using glue code, scripting or other late binding composition schemes in ecosystems, or other decentralized enterprises [24, 25], as a key facet that can enable software producers to build systems from diverse sources.

Jansen and colleagues [16, 17] observe that software ecosystems (a) embed software supply networks that span multiple organizations and (b) are embedded within a network of intersecting or overlapping software ecosystems that span the world of software engineering practice. Scacchi [30] for example, identifies that the world of open source software (OSS) development is a loosely coupled collection of software ecosystems different from those of commercial software producers and its supply networks are articulated within a network of FOSS development projects. Networks of OSS ecosystems have also begun to appear around very large OSS projects for Web browsers, Web servers, word processors, and others, as well as related application development environments like NetBeans and Eclipse, and these networks have become part of global information infrastructures [18].

OSS ecosystems also exhibit strong relationships between the ongoing evolution of OSS systems and their developer/user communities, such that the success of one co-depends on the success of the other [30]. Ven and Mannaert discuss the challenges independent software vendors face in combining OSS and proprietary components, with emphasis on how OSS components evolve and are maintained in this context [35].

Boucharas and colleagues [10] then draw attention to the need to more systematically and formally model the contours of software supply networks, ecosystems, and networks of ecosystems. Such a formal modeling base may then help in systematically reasoning about what kinds of relationships or strategies may arise within a software ecosystem. For example, Kuehnel [19] examines how Microsoft's software ecosystem developed around operating systems (MS *Windows*) and key applications (e.g., *Microsoft Office*) may be transforming from "predator" to "prey" in its effort to

control the expansion of its markets to accommodate OSS (as the extant prey) that eschew closed source software with proprietary software licenses.

Our work in this area builds on these efforts in the following ways. First, we share the view of a need for examining software ecosystems, but we start from software system architectures that can be formally modeled and analyzed with automated tool support [7, 32]. Explicit modeling of software architectures enables the ability to view and analyze them at design-time, build-time, or deployment/run-time. Software architectures also serve as a mechanism for coordinating decentralized software development across multi-site projects [28]. Similarly, explicit models allow for the specification of system architectures using either proprietary software components with open APIs, OSS components, or combinations thereof, thereby realizing open architecture (OA) systems [31]. We then find value in attributing open architecture components with their (intellectual property) licenses [2] because software licenses are an expression of contractual/social obligations that software consumers must fulfill in order to realize the rights to use the software in specified allowable manners, as determined by the software's producers.

3.3. Open Architectures

Open architecture (OA) software is a customization technique introduced by Oreizy [27] that enables third parties to modify a software system through its exposed architecture, evolving the system by replacing its components. Increasingly more software-intensive systems are developed using an OA strategy, not only with OSS components but also with proprietary components with open APIs (e.g. [33]). Using this approach can lower development costs and increase reliability and function [31]. Composing a system with heterogeneously-licensed components, however, increases the likelihood of conflicts, liabilities, and no-rights stemming from incompatible licenses. Thus, in our work we define an OA system as *a software system consisting of components that are either open source or proprietary with open API, whose overall system rights at a minimum allow its use and redistribution, in full or in part*.

It may appear that using a system architecture that incorporates OSS components and uses open APIs will result in an OA system. But not all such architectures will produce an OA because the (possibly empty) set of available license rights for an OA system depends on (a) how and why OSS and open APIs are located within the system architecture, (b) how OSS and open APIs are implemented, embedded, or interconnected, and (c) the degree to which the licenses of different

OSS components encumber all or part of a software system's architecture into which they are integrated [1, 31].

The following kinds of software elements appearing in common software architectures can affect whether the resulting systems are open or closed [6].

Software source code components—These can be either (a) standalone programs, (b) libraries, frameworks, or middleware, (c) inter-application script code such as C shell scripts, or (d) intra-application script code, such as for creating Rich Internet Applications using domain-specific languages such as XUL for the *Firefox* Web browser [14] or "mashups" [23]. Their source code is available and they can be rebuilt. Each may have its own distinct license, though often script code that merely connects programs and data flows does not, unless the code is substantial, reusable, or proprietary.

Executable components—These components are in binary form and the source code may not be open for access, review, modification, or possible redistribution [29]. If proprietary, they often cannot be redistributed, and so such components will be present in the design-, build-, and run-time architectures but not in the distribution-time architecture.

Software services—An appropriate software service can replace a source code or executable component.

Application programming interfaces/APIs—Availability of externally visible and accessible APIs is the minimum requirement for an "open system" [22]. Open APIs are not and cannot be licensed, but they can limit the propagation of license obligations.

Software connectors—Software whose intended purpose is to provide a standard or reusable way of communication through common interfaces, e.g. High Level Architecture [20], CORBA, MS .NET, Enterprise Java Beans, and GNU Lesser General Public License (LGPL) libraries. Connectors can also limit the propagation of license obligations.

Methods of composition—These include linking as part of a configured subsystem, dynamic linking, and client-server connections. Methods of composition affect license obligation propagation, with different methods affecting different licenses.

Configured system or subsystem architectures—These are software systems that are used as atomic components of a larger system, and whose internal architecture may comprise components with different licenses, affecting the overall system license. To minimize license interaction, a configured system or sub-architecture may be surrounded by what we term a license firewall, namely a layer of dynamic links, client-server connections, license shims, or other connectors that block the propagation of reciprocal obligations.

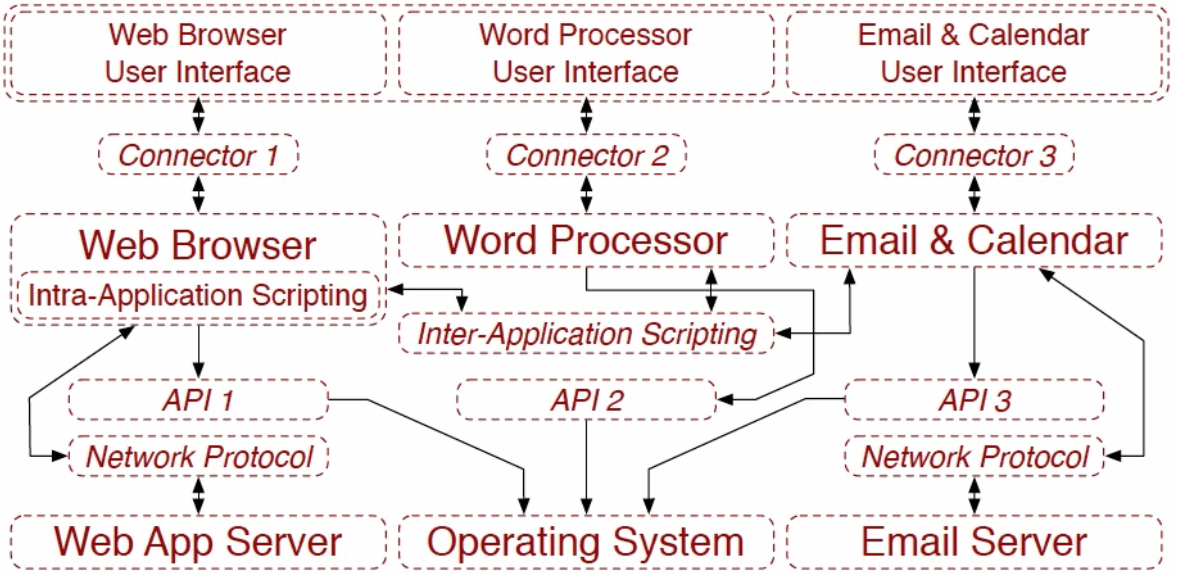


Figure 5: A design-time architecture

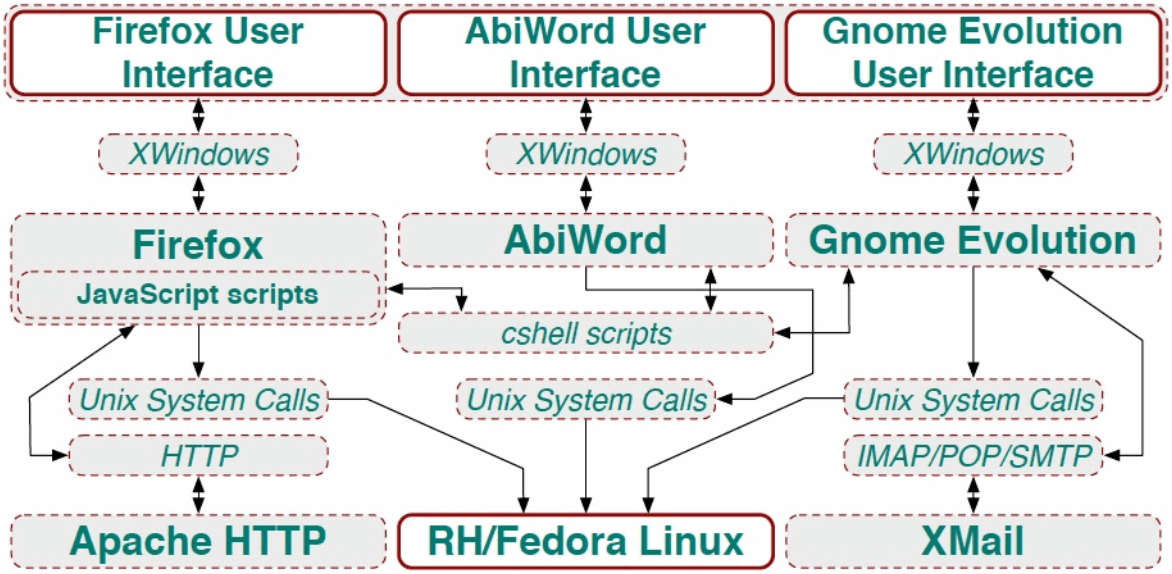


Figure 6: A build-time architecture

With these architectural elements, we can create a design-time or reference architecture for a system that conforms to the software supply network shown in Figure 2. This design-time architecture appears in Figure 5; note that it only specifies components by type rather than by producer, meaning the choice of producer component remains unbound at this point. Then in Figure 6, we create a build-time rendering of this architectural design by selecting specific components from designated software producers. The gray boxes correspond to components and connectors not visible in the run-time instantiation of the system in Figure 7.

Figures 7, 8, and 9 display alternative run-time instantiations of the design-time architecture of Figure 5. The architectural run-time instance in Figure 7 corresponds to the software ecosystem niche shown in Figure 3; Figure 8 corresponds to the niche in Figure 4; and Figure 9 designates yet another niche different from the previous two.

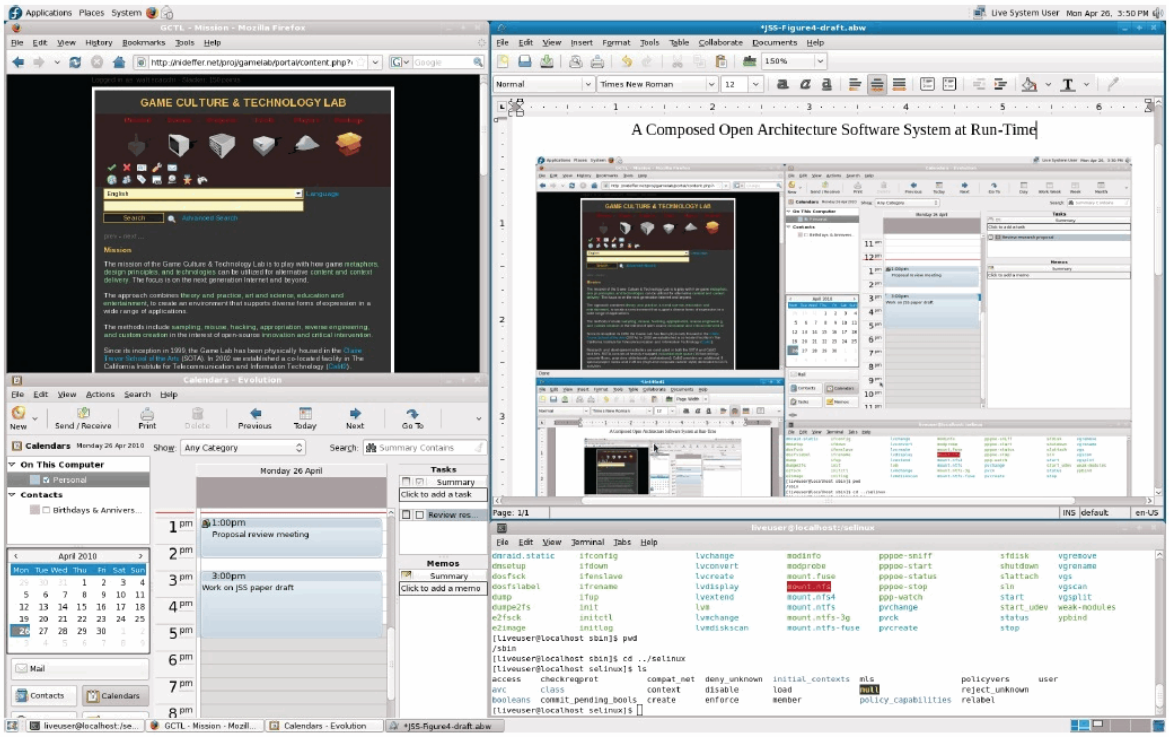


Figure 7: An instantiation at run-time (*Firefox, AbiWord, Gnome Evolution, Fedora*) of the build-time architecture of Figure 6 that determines the ecosystem niche of Figure 3

Each component selection implies acceptance of the license obligations and rights that the producer seeks to transmit to the components consumers. However, in an OA design development, component interconnections may be used to intentionally or unintentionally propagate these obligations onto other components whose licenses may conflict with them or fail to match [2, 15]; the system integrator can decide to insert software shims using scripts, dynamic links to remote services, data communication protocols, or libraries to mitigate or fire-wall the extent to which a component's license obligations propagate. This style of build-time composition can be used to accommodate a system's consumers' policy for choosing systems that avoid certain licenses or that isolate the license obligations of certain desirable components. It also allows system integrators and consumers to follow a "best of breed" policy in the selection of system components. Finally, if no license conflicts exist in the system, or if the integrator and system consumer are satisfied with the component and license choices made, then the compositional bindings may simply be set in the most efficient way available. This realizes a policy for accepting only components and licenses whose obligations and rights are acceptable to the system consumers.

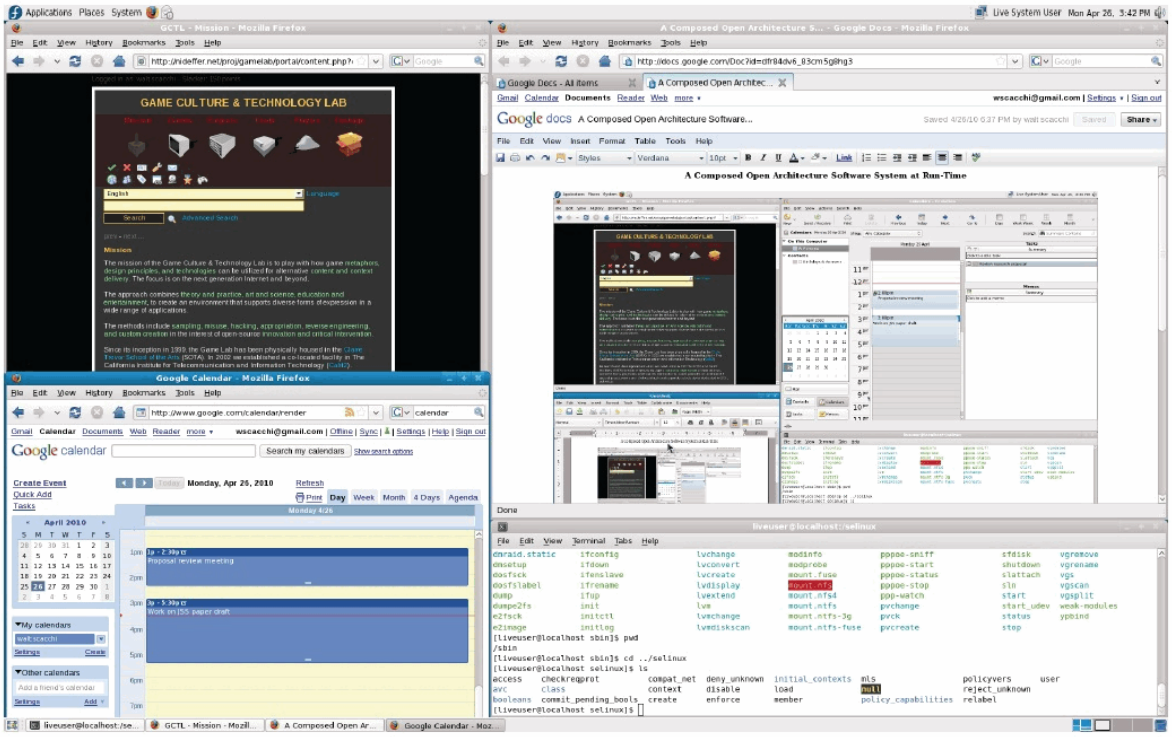


Figure 8: A second instantiation at run-time (*Firefox, Google Docs and Calendar, Fedora*) determining the ecosystem niche of Figure 4

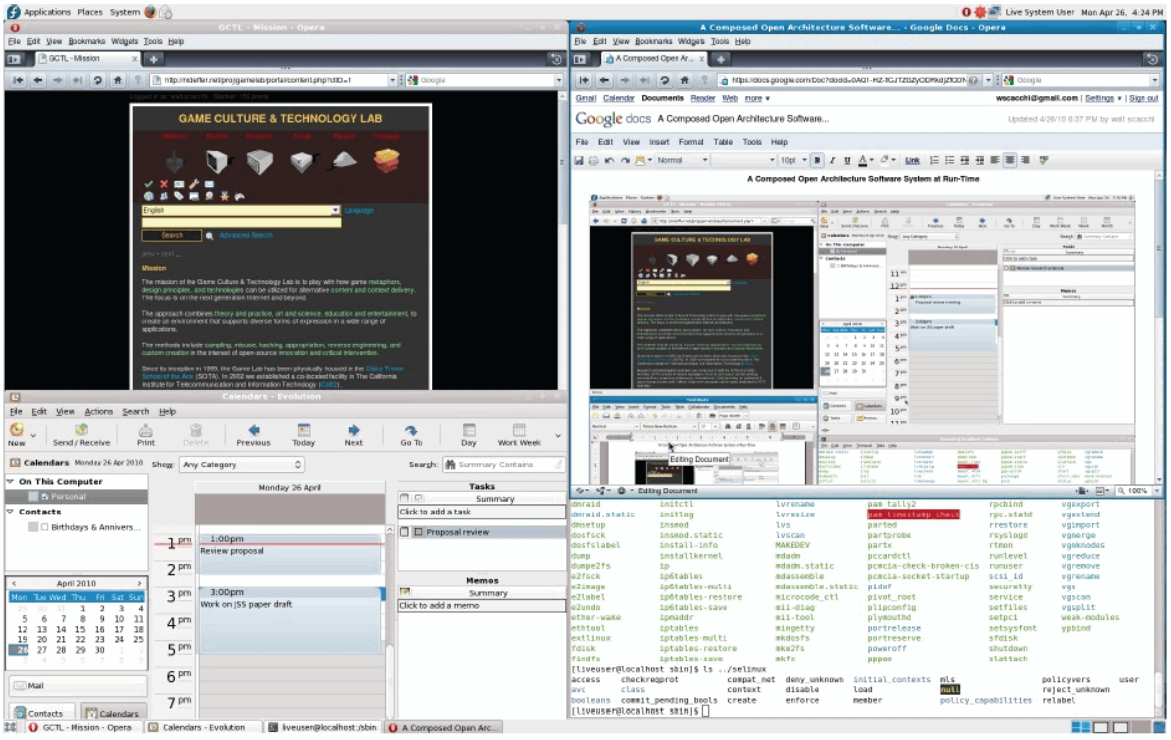


Figure 9: A third instantiation at run-time (*Opera, Google Docs, Gnome Evolution, Fedora*) determining yet another niche conforming to the software supply network of Figure 2

3.4. Software Licenses

Traditional proprietary licenses allow a company to retain control of software it produces, and restrict the access and rights that outsiders can have. OSS licenses, on the other hand, are designed to encourage sharing and reuse of software, and grant access and as many rights as possible. OSS licenses are classified as *academic* or *reciprocal*. Academic OSS licenses such as the Berkeley Software Distribution (BSD) license, the Massachusetts Institute of Technology license, the Apache Software License, and the Artistic License, grant nearly all rights to components and their source code, and impose few obligations. Anyone can use the software, create derivative works from it, or include it in proprietary projects. Typical academic obligations are simply to not remove the copyright and license notices.

Reciprocal OSS licenses take a more active stance towards sharing and reusing software by imposing the obligation that reciprocally-licensed software not be combined (for various definitions of “combined”) with any software that is not in turn also released under the reciprocal license. Those for which most or all ways of combining software propagate reciprocal obligations are termed *strongly reciprocal*. Examples are the GPL and the Affero GPL (AGPL). The purpose of the AGPL is to prevent a GPL software component from being integrated into an OA system as a remote server, or from being wrapped with shims to inhibit its ability to propagate the GPL obligations and rights. The purpose of these licenses is to ensure that software so licensed will maintain (and can propagate) the freedom to access, study, modify, and redistribute the software source code, which academic licenses do not. This in turn assures the access, use, and reusability of the source code for other software producers and system integrators. Those licenses for which only certain ways of combining software propagate reciprocal obligations are termed *weakly reciprocal*. Examples are the Lesser GPL (LGPL), Mozilla Public License (MPL), and Common Public License. The goals of reciprocal licensing are to increase the domain of OSS by encouraging developers to bring more components under its aegis and to prevent improvements to OSS components from vanishing behind proprietary licenses.

Both proprietary and OSS licenses typically disclaim liability, assert no warranty is implied, and obligate licensees to not use the licensor’s name or trademarks. Newer licenses often cover patent issues as well, either giving a restricted patent license or explicitly excluding patent rights. The Mozilla Disjunctive Tri-License licenses the core Mozilla components under any one of three licenses (MPL, GPL, or the GNU Lesser General Public License LGPL); OSS developers or OA

system integrators can choose the one that best suits their needs for a particular project and component.

The Open Source Initiative (OSI) maintains a widely-respected definition of “open source” and gives its approval to licenses that meet it [26]. OSI maintains and publishes a repository of approximately 70 approved OSS licenses which tend to vary in the terms and conditions of their declared obligations and rights. However, all of these licenses tend to cluster into either a strongly reciprocal, weakly reciprocal, or minimally restrictive/academic license type.

Common practice has been for an OSS project to choose a single license under which all of its products are released and to require developers to contribute their work only under conditions compatible with that license. For example, the Apache Contributor License Agreement grants enough of each author’s rights to the Apache Software Foundation for the foundation to license the resulting systems under the Apache Software License. This sort of rights regime, in which the rights to a system’s components are homogeneously granted and the system has a single well-defined OSS license, was the norm in the early days of OSS and continues to be practiced.

We have developed an approach for expressing software licenses that is more formal and less ambiguous than natural language, and that allows us to calculate and identify conflicts arising from the rights and obligations of two or more components’ licenses. Our approach is based on Hohfeld’s classic group of eight fundamental jural relations [Hohfeld 1913], of which we use right, duty, no-right, and privilege (Figure 10). We start with a tuple <actor, operation, action, object> for expressing a right or obligation. The actor is the “licensee” for all of the licenses we have examined. The operation is one of the following: “may”, “must”, “must not”, or “need not”, with “may” and “need not” expressing rights and “must” and “must not” expressing obligations. Because copyright rights are only available to entities who have been granted a sub-license, only the listed rights are available, and the absence of a right means that it is not available. The action is a verb or verb phrase describing what may, must, must not, or need not be done, with the object completing the description. A license may be expressed as a set of rights, with each right associated with zero or more obligations that must be fulfilled in order to enjoy that right. Figure 11 sketches two rights from GPL 2.0, the first one with no obligations and the second with three corresponding obligations. Elsewhere, the details of this license specification scheme, how it can be used to attribute software architectures to produce a system license architecture, and how it can be formalized into a semantic meta-model [2].

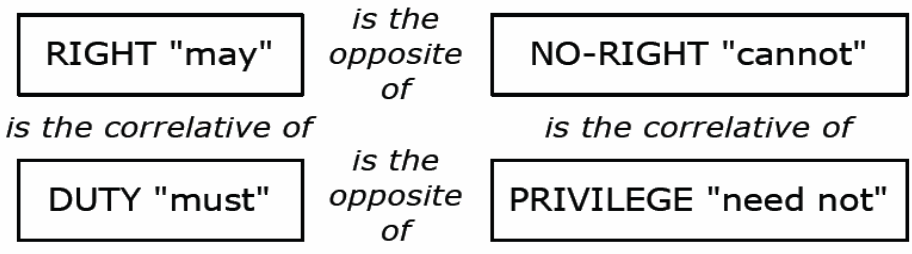


Figure 10: Hohfeld's four basic relations

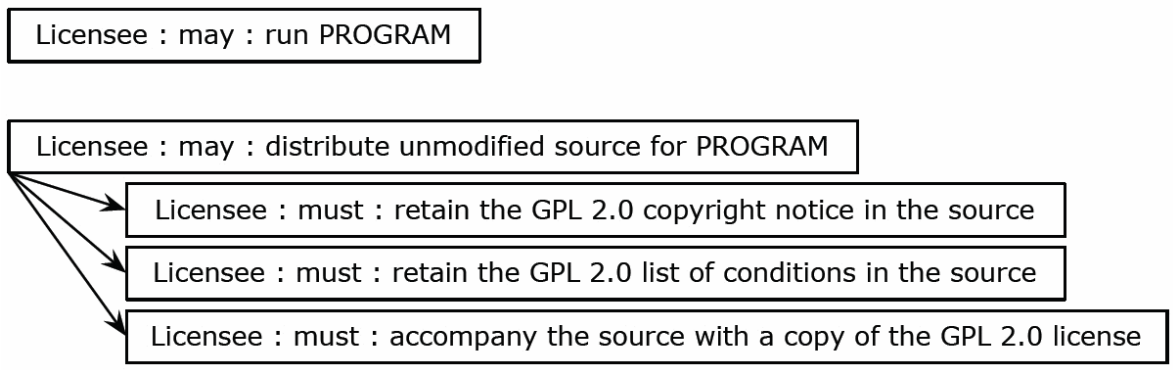


Figure 11: Tuples for some rights and obligations of the GPL 2.0 license

HLS designers have developed a number heuristics to guide architectural design while avoiding some license conflicts. First, it is possible to use a reciprocally-licensed component through a license firewall that limits the scope of reciprocal obligations. Rather than connecting conflicting components directly through static or other build-time links, the connection is made through a dynamic link, client-server protocol, license shim (such as an LGPL connector), or run-time plug-ins. A second approach used by a number of large organizations is simply to avoid using any components with reciprocal licenses. A third approach is to meet the license obligations (if that is possible) by, for example, retaining copyright and license notices in the source and publishing the source code. However, even using design heuristics such as these (and there are many), keeping track of license rights and obligations across components that are interconnected in complex OAs quickly becomes too cumbersome. Automated support is needed to manage the multi-component, multi-license complexity. Accordingly, we are developing an automated support capability as part of the *ArchStudio* architecture design environment [4, 5, 13] that can analyze the addition of software license properties such as those shown in Figure 11 to the interfaces of software components in an OA system. For example, in Figure 12 we see a rendering of the OA system from our case study with the *AbiWord* word processing component highlighted. This component's APIs would be attributed with the GPL license obligations and rights in Figure 11 because *AbiWord* is licensed under GPL, as are other components like the Gnome *Evolution* calendaring and email application

and also the Red Hat/*Fedora Linux* operating system platform. Because the architectural interconnections shown in the model of Figure 12 indicate that none of these components covered by GPL are directly interconnected to another licensed component, their license obligations do not propagate or become “viral” in this architectural composition. Replacing any of these GPL components with non-GPL but still OSS components would not change the total set of obligations and rights on the system with this architecture; the system would remain OSS, but the software ecosystem niche in which it resides would shift to another niche. Once again, software licenses interact with software architectures and together they help determine which software ecosystem

niche will embed an instantiated run-time version of the system.

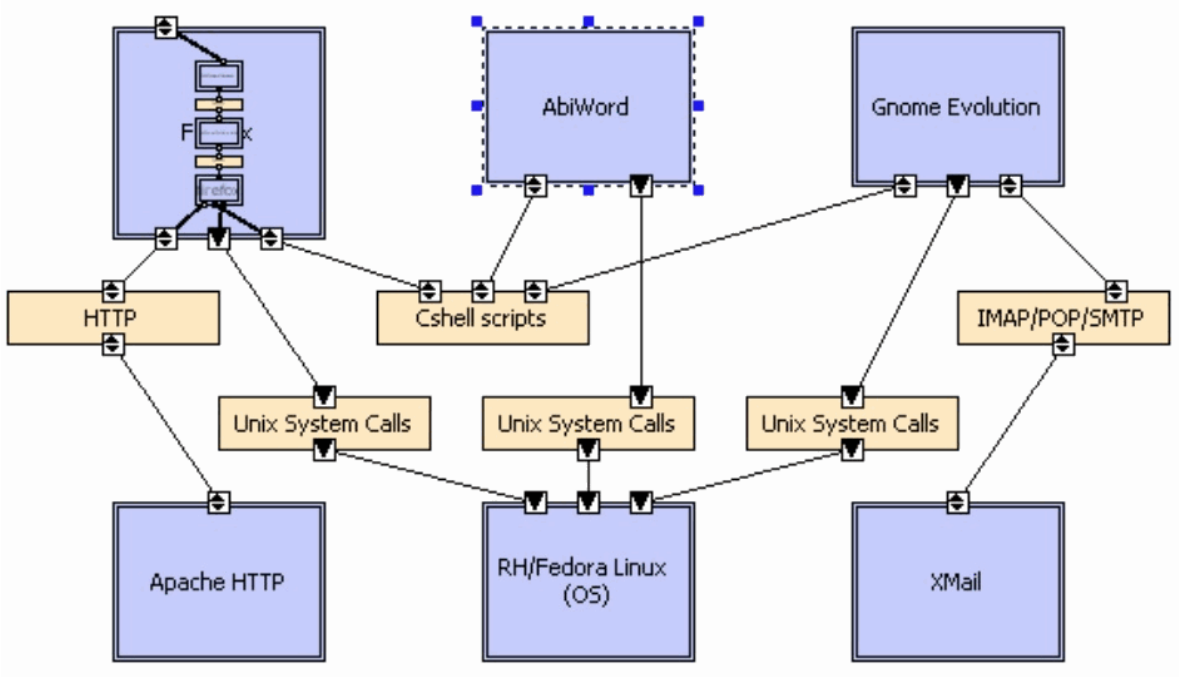


Figure 12: The model of the system architecture used in our case study as rendered in our automated tool, described elsewhere [4, 2]

3.5. Architecture, License, and Ecosystem Evolution

An OA system can evolve by a number of distinct mechanisms, some of which are common to all systems but others of which are a result of heterogeneous component licenses in a single system.

By component evolution—One or more components can evolve, altering the overall system’s characteristics (for example, upgrading and replacing the *Firefox* Web browser from version 3.5 to 3.6). Such minor version changes generally have no effect on system architecture.

By component replacement—One or more components may be replaced by others with modestly different functionality but similar interface, or with a different interface and the addition of shim code to make it match (for example, replacing the *AbiWord* word processor with either *OpenOffice Writer* or *Microsoft Word*). However, changes in the format or structure of component APIs may necessitate build-time and run-time updates to component connectors. Figure 13 shows some possible alternative system compositions that result from replacing components by others of the same type but with a different license.

By architecture evolution—The OA can evolve, using the same components but in a different configuration, altering the system characteristics. For example, as discussed in Section 4, revising or refactoring the configuration in which a component is connected can change how its license affects the rights and obligations for the overall system. This could arise when replacing word processing, calendaring, and email components and their connectors with Web browser-based services such as *Google Docs*, *Google Calendar*, and *Google Mail*. The replacement would eliminate the legacy components and relocate the desired application functionality to operate within the Web browser component, resulting in what might be considered a simpler and easier-to-maintain system architecture, but one that is less open and now subject to a proprietary Terms of Service license. System consumer policy preferences for licenses, and subsequent participation in a different ecosystem niche, may thus mediate whether such an alternative system architecture is desirable or not.

By component license evolution—The license under which a component is available may change, as for example when the license for the Mozilla core components was changed from the Mozilla Public License (MPL) to the current Mozilla Disjunctive Tri-License; or the component may be made available under a new version of the same license, as for example when the GNU General Public License (GPL) version 3 was released. The three architectures in Figure 13 that incorporate the Firefox Web browser show how its tri-license creates new evolutionary paths by offering different licensing options. These options and paths were not available previously with earlier versions of this component offered under only one or two license alternatives.

By a change to the desired rights or acceptable obligations—The OA system's integrator or consumers may desire additional license rights (for example the right to sublicense in addition to the right to distribute), no longer desire specific rights, or the set of license obligations they find

acceptable may change. In either case the OA system evolves, whether by changing components, evolving the architecture, or other means, to provide the desired rights within the scope of the acceptable obligations. For example, they may no longer be willing or able to provide the source code for components within the reciprocality scope of a GPL-licensed module. Figure 14 shows an array of choices among types of licenses for different types of components that appear in the OA case study example. Each choice determines the obligations that component producers can

demand of their consumers in exchange for the access/usage rights they offer.

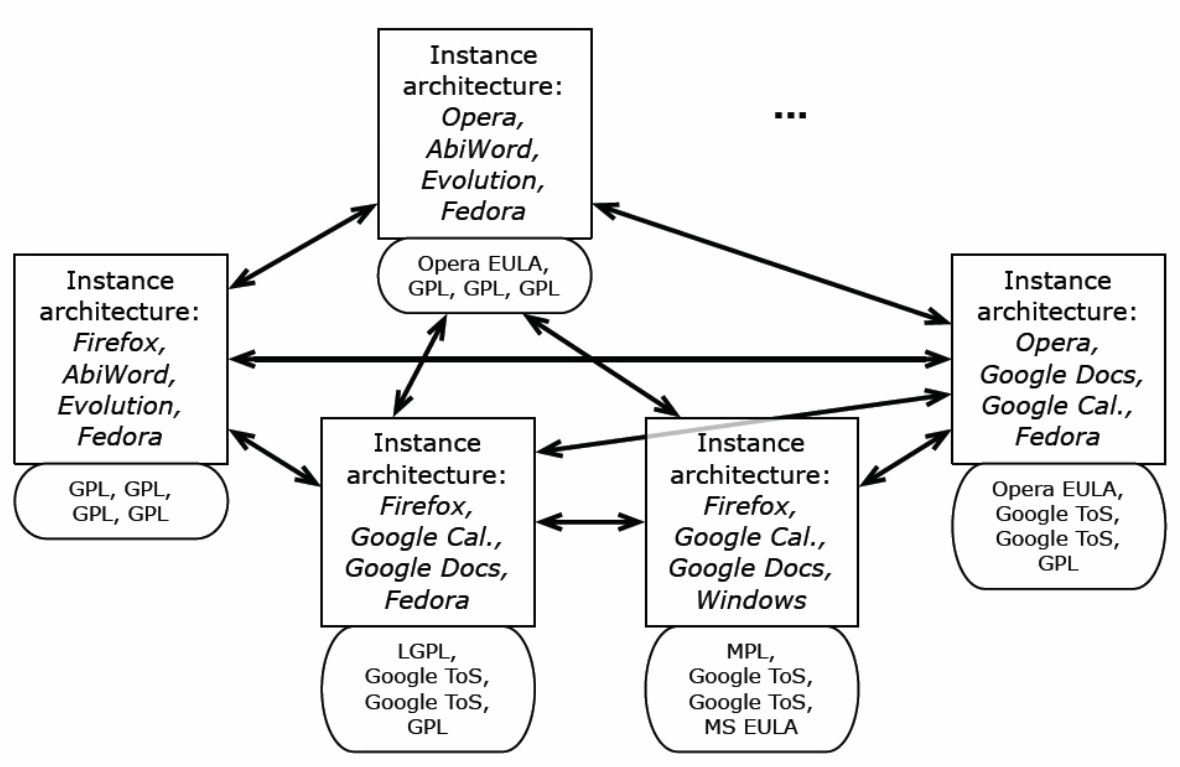


Figure 13: Possible evolutionary paths among a few instance architectures; some paths are impractical due to the changes in license obligations

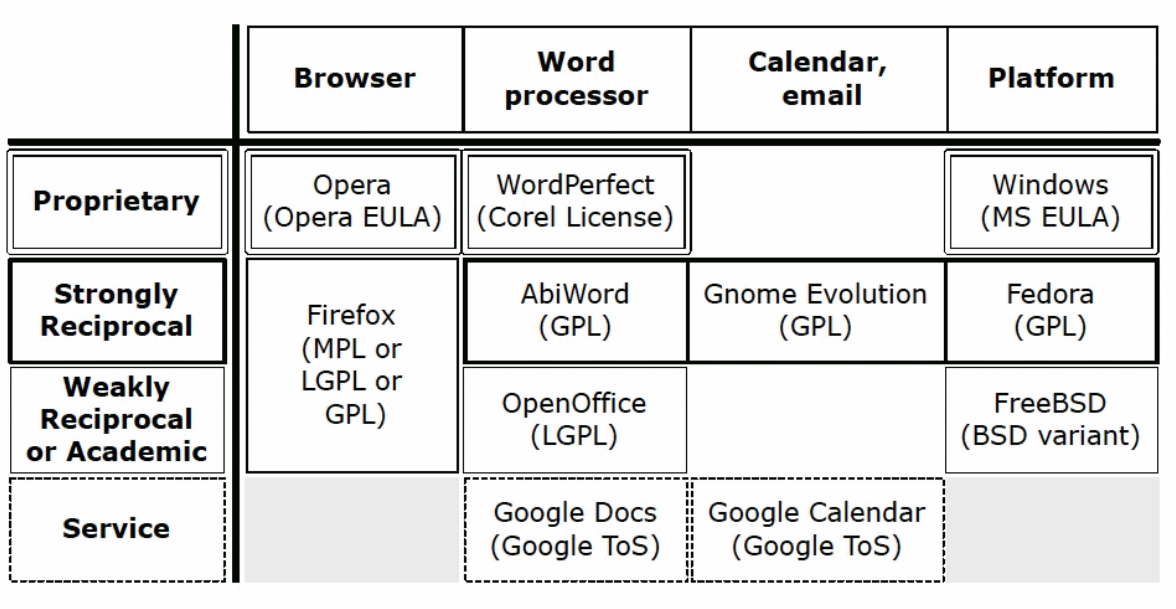


Figure 14: Some architecture choices and their license categories

3.6. Discussion

At least two topics merit discussion following from our approach to the understanding of software ecosystems and ecosystem niches for OA systems: first, how might our results shed light on software systems whose architectures articulate a software product line and second, what insights might we gain based on the results presented here on possible software license architectures for mobile computing ecosystems. Each is addressed in turn. Software product lines (SPLs) rely on the development and use of explicit software architectures [7, 12]. However, the architecture of an SPL or software ecosystem does not necessarily require an OA—there is no need for it to be open. Thus, we are interested in discussing what happens when SPLs may conform to an OA, and to an OA that may be subject to heterogeneously licensed SPL components. Three considerations come to mind:

1. If the SPL is subject to a single homogeneous software license, which may often be the case when a single vendor or government contractor has developed the SPL, then the license may act to reinforce a vendor lock-in situation with its customers. One of the motivating factors for OA is the desire to avoid such lock-in, whether or not the SPL components have open or standards-compliant APIs. However, a single license simplifies determination of the software ecosystem in which the system is located.

2. If an OA system employs a reference architecture, then such a reference or design-time architecture effectively defines an SPL consisting of possible different system instantiations composed from similar components from different producers (e.g., different but equivalent Web browsers, word processors, calendaring, and email applications). This can be seen in the design-time architecture depicted in Figure 5, the build-time architecture in Figure 6, and the instantiated run-time architectures in Figures 7, 8, and 9.
3. If the SPL is based on an OA that integrates software components from multiple producers or OSS components that are subject to different heterogeneous licenses, then we have the situation analogous to what we have presented in this paper, but now in the form of virtual SPLs from a virtual software production enterprise [24] that spans multiple independent OSS projects and software production enterprises. As such, SPL concepts are compatible with OA systems that are composed from heterogeneously licensed components, but do not impact the formation or evolution of the software ecosystem niches where such systems may reside.

Our approach for using open software system architectures and component licenses as a lens that focuses attention to certain kinds of relationships within and across software supply networks, software ecosystems, and networks of software ecosystems has yet to be applied to systems on mobile computing platforms. Bosch [8] notes that this is a neglected area of study but one that may offer interesting opportunities for research and software product development. Thus, what happens when we consider Apple *iPhone/iPad* operating system (OS), Google *Android* OS phones, Nokia *Symbian* OS phones, Nokia *Maemo* OS hand-held computers, Microsoft *Windows 7* OS phones, Intel *Moblin* OS netbooks, or *Nintendo DS* portable game consoles as possible platforms for OA system design and deployment?

- All of these devices are just personal computers with operating systems, albeit in small, easy to carry, and wireless form factors. They represent a mix of mostly proprietary operating system platforms, though some employ Linux-based or other OSS alternative operating systems.
- Mobile OS platforms owners (Apple, Nokia, Google, Microsoft) are all acting to control the software ecosystems for consumers of their devices through establishment of logically centralized (but possibly physically decentralized) application distribution repositories or online stores, where the mobile device must invoke a networked link to the repository to acquire (for fee/free) and install apps. Apple has had the greatest success in this strategy and dominates the global mobile application market and mobile computing software

ecosystems. But overall, OA systems are not necessarily excluded from these markets or consumers.

- Given our design-time architecture of the case study system shown in Figure 5, is it possible to identify a build-time version that could produce a run-time version that could be deployed on most or all of these mobile devices? One such build-time architecture would compose an *Opera* Web browser with Web services for word processing, calendaring, and email that could be hosted on either proprietary or OSS mobile operating systems. This alternative arises because Opera Software has produced run-time versions of its proprietary Web browser for these mobile operating systems for accessing the Web via a wireless/cellular phone network connection. Similarly, in Figure 13 the instance architecture on the right could evolve to operate on a mobile platform like an *Android*-based mobile device or *Symbian*-based cell phone. So it appears that mobile computing devices do not pose any unusual challenges for our approach in terms of understanding their software ecosystems or the ecosystem niches for OA systems that could be hosted on such devices.

3.7. Conclusion

The role of software ecosystems in the development and evolution of heterogeneously-licensed open architecture systems has received insufficient consideration. Such systems are composed of components potentially under two or more licenses, open source software or proprietary or both, in an architecture in which evolution can occur by evolving existing components, replacing them, or refactoring. The software licenses of the components both facilitate and constrain in which ecosystems a composed system may lie. In addition, the obligations and rights carried by the licenses are transmitted from the software component producers to system consumers through the architectural choices made by system integrators. Thus, software component licenses help determine the contours of the software supply network and software ecosystem niche that emerge for a given implementation of a composed system architecture. Accordingly, we described examples for systems whose host software platform span the range of personal computer operating systems, Web services, and mobile computing devices.

Consequently, software component licenses and the architectural composition of a system determine the software ecosystem niche where a system lies. Understanding and describing software ecosystem niches is a key contribution of this work. A case study of an open architecture software system that articulates different niches was employed to this end. We examined how the architecture and software component licenses of a composed system at design-time, build-time,

and run-time help determine the system's software ecosystem niche, and provide insight for identifying potential evolutionary paths of software system, architecture, and niches. Similarly, we detailed the ways in which a composed system can evolve over time, and how a software system's evolution can change or shift the software ecosystem niche in which the system resides and thus producer-consumers relationships. Then we described how virtual software product lines can be identified through a lens that examines the association between open architectures, software component licenses, and software ecosystems.

Finally, in related work [4, 2, 3] we identified structures for modeling software licenses and the license architecture of a system and automated support for calculating its rights and obligations. Such capabilities are needed in order to manage and track an OA system's evolution in the context of its ecosystem niche. We have outlined an approach for achieving these structures and support and sketched how they further the goal of reusing and exchanging alternative software components and software architectural compositions. More work remains to be done, but we believe this approach transforms a vexing problem of stating in detail how the study of software ecosystems can be tied to core issues in software engineering like software architecture, product lines, component-based reuse, license management, and evolution into a manageable one for which workable solutions can be obtained.

Acknowledgments

This research is supported by grants #0808783 from the U.S. National Science Foundation, and grant #N00244-10-1-0038 from the Acquisition Research Program at the Naval Postgraduate School. No review, approval, or endorsement is implied.

3.8. References

[1] Alspaugh, T. A., Anton, A. I., 2008. Scenario support for effective requirements. Information and Software Technology 50 (3), 198–220.

[2] Alspaugh, T. A., Asuncion, H. U., Scacchi, W., 2009. Intellectual property rights requirements for heterogeneously-licensed systems. In: 17th IEEE International Requirements Engineering Conference (RE'09). pp. 24–33.

[3] Alspaugh, T. A., Asuncion, H. U., Scacchi, W., 2009. The role of software licenses in open architecture ecosystems. In: First International Workshop on Software Ecosystems (IWSECO-2009). pp. 4–18.

[4] Alspaugh, T. A., Scacchi, W., Asuncion, H. U., Nov. 2010. Software licenses in context: The challenge of heterogeneously-licensed systems. Journal of the Association for Information Systems 11 (11), 730–755.

[5] Asuncion, H. U., 2009. Architecture-centric traceability for stakeholders (ACTS). Ph.D. thesis, University of California, Irvine.

[6] Bass, L., Clements, P., Kazman, R., 2003. Software Architecture in Practice. Addison-Wesley Longman.

[7] Bosch, J., 2000. Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach. Addison-Wesley.

[8] Bosch, J., 2009. From software product lines to software ecosystems. In: 13th International Software Product Line Conference (SPLC'09). pp. 111–119.

[9] Bosch, J., Bosch-Sijtsema, P., 2010. From integration to composition: On the impact of software product lines, global development and ecosystems. J. Systems and Software, 83 (1), 67–76.

[10] Boucharas, V., Jansen, S., Brinkkemper, S., 2009. Formalizing software ecosystem modeling. In: First International Workshop on Open Component Ecosystems (IWOCE'09). pp. 41–50.

[11] Brown, A. W., Booch, G., 2002. Reusing open-source software and practices: The impact of open-source on commercial vendors. In: Software Reuse: Methods, Techniques, and Tools (ICSR-7). pp. 381–428.

[12] Clements, P., Northrop, L., 2001. Software Product Lines: Practices and Patterns. Addison-Wesley Professional.

[13] Dashofy, E. M., 2007. Supporting stakeholder-driven, multi-view software architecture modeling. Ph.D. thesis, University of California, Irvine.

[14] Feldt, K., 2007. Programming Firefox: Building Rich Internet Applications with XUL. O'Reilly

Media, Inc.

[15] German, D. M., Hassan, A. E., 2009. License integration patterns: Dealing with licenses mismatches in component-based development. In: 28th International Conference on Software Engineering (ICSE '09). pp. 188–198.

[16] Jansen, S., Brinkkemper, S., Finkelstein, A., 2009. Business network management as a survival strategy: A tale of two software ecosystems. In: First Workshop on Software Ecosystems. pp. 34–48.

[17] Jansen, S., Finkelstein, A., Brinkkemper, S., 2009. A sense of community: A research agenda for software ecosystems. In: 28th International Conference on Software Engineering (ICSE '09), Companion Volume. pp. 187–190.

[18] Jensen, C., Scacchi, W., Jul./Sep. 2005. Process modeling across the web information infrastructure. Software Process: Improvement and Practice 10 (3), 255–272.

[19] Kuehnel, A.-K., Jun. 2008. Microsoft, open source and the software ecosystem: of predators and prey—the leopard can change its spots. Information & Communucation Technology Law, 17(2), 107–124.

[20] Kuhl, F., Weatherly, R., Dahmann, J., 1999. Creating computer simulation systems: an introduction to the high level architecture. Prentice Hall.

[21] Messerschmitt, D. G., Szyperski, C., 2003. Software Ecosystem: Understanding an Indispensable Technology and Industry. MIT Press.

[22] Meyers, B. C., Oberndorf, P., 2001. Managing Software Acquisition: Open Systems and COTS Products. Addison-Wesley Professional.

[23] Nelson, L., Churchill, E. F., 2006. Repurposing: Techniques for reuse and integration of interactive systems. In: International Conference on Information Reuse and Integration (IRI-08). p. 490.

[24] Noll, J., Scacchi, W., Feb. 1999. Supporting software development in virtual enterprises. J. of Digital Information 1(4).

[25] Noll, J., Scacchi, W., 2001. Specifying process-oriented hypertext for organizational computing. J. Network and Computing Applications 24 (1), 39–61.

[26] Open Source Initiative, 2010. Open Source Definition.
http://www.opensource.org/docs/osd.

[27] Oreizy, P., 2000. Open architecture software: A flexible approach to decentralized software evolution. Ph.D. thesis, University of California, Irvine.

[28] Ovaska, P., Rossi, M., Marttiin, P., 2003. Architecture as a coordination tool in multi-site software development. Software Process: Improvement and Practice 8 (4), 233–247.

[29] Rosen, L., 2005. Open Source Licensing: Software Freedom and Intellectual Property Law. Prentice Hall.

[30] Scacchi, W., 2007. Free/open source software development: Recent research results and emerging opportunities. In: 6th Joint European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2007). pp. 459–468.

[31] Scacchi, W., Alspaugh, T. A., May 2008. Emerging issues in the acquisition of open source software within the U.S. Department of Defense. In: 5th Annual Acquisition Research Symposium, Monterey, CA

[32] Taylor, R. N., Medvidovic, N., Dashofy, E. M., 2009. Software Architecture: Foundations, Theory, and Practice. Wiley.

[33] Unity Technologies, Dec. 2008. End User License Agreement. http://unity3d.com/unity/unity-end-user-license-2.x.html .

[34] van Gurp, J., Prehofer, C., Bosch, J., 2010. Comparing practices for reuse in integration-oriented software product lines and large open source software projects. Software — Practice & Experience, 40 (4), 285–312.

[35] Ven, K., Mannaert, H., 2008. Challenges and strategies in the use of open source software by independent software vendors. Information and Software Technology 50 (9-10), 991–1002.

THIS PAGE INTENTIONALLY LEFT BLANK

4. Intellectual Property Rights Requirements for Heterogeneously-Licensed Systems

Thomas A. Alspaugh
Computer Science Dept., Georgetown University, Washington, DC, USA

Hazeline U. Asuncion and Walt Scacchi
Institute for Software Research. University of California, Irvine USA

September 2009

Abstract

Heterogeneously-licensed systems pose new challenges to analysts and system architects. Appropriate intellectual property rights must be available for the installed system, but without unnecessarily restricting other requirements, the system architecture, and the choice of components both initially and as it evolves. Such systems are increasingly common and important in e-business, game development, and other domains. Our semantic parameterization analysis of open-source licenses confirms that while most licenses present few roadblocks, reciprocal licenses such as the GNU General Public License produce knotty constraints that cannot be effectively managed without analysis of the system's *license architecture*. Our automated tool supports intellectual property requirements management and license architecture evolution. We validate our approach on an existing heterogeneously-licensed system.

Published as: T.A. Alspaugh, H.A. Asuncion, and W. Scacchi, Intellectual Property Rights Requirements for Heterogeneously Licensed Systems, in *Proc. 17th. Intern. Conf. Requirements Engineering (RE09*), Atlanta, GA, 24-33, September 2009.

4.1. Introduction

Until recently, the norm for licensed software has been that software is used and distributed under the terms of a single license, with all of its components homogeneously licensed under a single proprietary or open source software (OSS) license. It is increasingly common to see heterogeneously-licensed systems (HLSs), whose components are not under the same license [8, 19, 21]. For web systems this has become so common that commercial tools for creating such "mashups" have been available for several years [9, 12]. Carefully constrained design, possibly aided by license exceptions from the copyright owners, may enable the resulting system to have a single specific license [8]. Otherwise the system as a whole has no single license, but rather one or more rights that are the intersection of all the component license's rights, and the union of their obligations. An example of an HLS is the Unity game development tool, whose license agreement lists eleven distinct licenses for its components, in addition to its overall license terms granting the right to use the system [21].

The intellectual property (IP) in a system—copyrights, patents, trademarks, trade dress, and trade secrets—is protected and made available through the licenses of the system and its components. IP requirements are expressed in terms of these licenses and the rights and obligations they entail, and include:

- the right to use, distribute, sub-license, etc.;
- the component selection strategy (whether limited to specific licenses, or open to "best-of-breed");
- interoperation of systems with specific IP regimes;
- the extent to which the system will be an open architecture (OA); and
- how it is distributed to, constituted by, and (for OA systems) evolved by users.

The IP requirements interact with the system's design-time, distribution-time, and run-time architectures in distinct ways, with the possibility of rights that conflict with other licenses' obligations, obligations that conflict across licenses, and unobtainable rights. The result can be a system that can't legally be sub-licensed, distributed, or used, or that involves its developers, distributors, or users in legal liabilities. Of course, some will ignore these legal issues (and anecdotal evidence indicates that many do), but companies and governments cannot afford to. Source code scanning services provided by third-party vendors address only one after-the-fact

aspect of this problem. While heuristics exist for managing IP requirements and are used in HLS development practice, they impose costs, unnecessarily limit the design space, and can result in a suboptimal, unsatisfactory system.

Software licenses and IP rights represent a new class of nonfunctional requirements, and constrain the development of systems with open architectures.

As part of our ongoing investigation of OSS and OA systems, we performed a grounded theory, semantic parameterization analysis of nine OSS licenses [4]. From this analysis and related work on OSS licensing [7, 18, 20], we were able to produce a meta-model for software licenses and for the contexts in which they are applied, and a calculus for license rights and obligations in license and context models. Using them, we calculate rights and obligations for specific systems, identify conflicts and unsupported rights, and evaluate alternative architectures and components and guide choices. We argue that these calculations are needed in developing systems of components whose licenses can conflict, whose design-time, distribution-time, and run-time architectures are not identical, whose component licenses may change through evolution, or for which a "best-of-breed" component strategy is desired. We have validated the approach by encoding the copyright rights and obligations for a group of OSS and proprietary licenses, implementing an architecture tool to automate calculations on the licenses, and applying it to an OSS-OA reference architecture. These models bring into clear relief the knotty constraints produced by interactions among proprietary licenses and reciprocal licenses such as the GNU General Public License (GPL), Mozilla Public License (MPL), and IBM's Common Public License (CPL). We have also discovered a novel second possible mechanism of interaction, through sources shared among compiled components under different licenses.

The main contributions of this work are the concept of a license firewall (Section 3.3), a meta-model for software licenses (Section 5), the concept of a license architecture (Section 5), an analysis process for determining the rights available for a system and their corresponding obligations (Section 6), an implementation of this analysis in an architecture development environment (Section 7), and the concept of a virtual license (Section 8).

The remainder of the paper is organized as follows. Section 2 outlines a motivating example from our own experience. Section 3 gives background. Related work is in Section 4. We discuss our analysis and meta-model of OSS licenses in Section 5, and the system contexts and calculations

on them in Section 6. Section 7 presents our tool support and its application to the reference model. We discuss implications in Section 8 and conclude in Section 9.

4.2. A motivating example

Heterogeneous software licenses can limit architectural choices when building and distributing multi-component systems, as illustrated by our recent experience prototyping a new multimedia content management portal that included support for videoconferencing and video recording and publishing. Our prototype was based on an Adobe Flash Media Server (FMS), and we developed both broadcast and multi-cast clients for video and audio that shared their data streams through the FMS. FMS is a closed source media server whose number of concurrent client connections is limited by a license fee. As the FMS license did not allow for redistribution, we could invite remote users to try out our clients and media services, but we could not offer to share the run-time environment that included the FMS. We could distribute our locally-developed clients and service source code. However, other potential developers at remote locations would then need to download and install a licensed copy of the FMS and then somehow rebuild our system using the source code we provided and their local copy of the FMS. In our view, this created a barrier to sharing the emerging results from our prototyping effort. We subsequently undertook to replace the FMS with Red5, an open source Flash media server, so we could distribute a run-time version of our content management portal to remote developers. Now these developers could install and use our run-time system, or download the source code, build, and share their own run-time version. Our experience shows how common software R&D efforts can be hampered in surprising ways by software components whose heterogeneous licenses limit distribution and sharing of work in progress.

4.3. Background

4.3.1. Intellectual Property (IP) basics

An individual can own a tangible thing and have property rights in it such as the rights to use it, improve it, sell it or give it away, or prevent others from doing so, subject to some statutory restrictions. Similarly, an individual can own intellectual property (IP) of various types and have specific property rights in the intangible intellectual property, such as the rights to copy, use, change, distribute, or prevent others from doing so, again subject to some statutory restrictions. In

the United States and most other countries, intellectual property is defined by

- copyright for a specific original expression of an idea,
- patent for an invention,
- trademark for a symbol, image, or phrase identifying the origin of products,
- trade dress for distinctive product packaging, and
- trade secret for an idea kept confidential.

Software licenses are primarily concerned with copyrights and patents and mention trademarks only to restrict a licensee's use of them; licenses rarely discuss trade dress or trade secrets [18]. In this paper we focus on copyright aspects of licenses.

Copyright is defined by Title 17 of the U.S. Code and by similar law in many other countries. It grants exclusive rights to the author of an original work in any tangible means of expression, namely the rights to reproduce the copyrighted work, prepare derivative works, distribute copies, and (for certain kinds of work) perform or display it. Because the rights are exclusive, the author can prevent others from exercising them, except as allowed by "fair use". The author can also grant others any or all of the rights or any part of them; one of the functions of a software license is to grant such rights and to define the conditions under which they are granted.

Copyright subsists in the expression of the original work, that is, the rights begin from the moment the work is expressed. In the U.S., a copyright lasts for the author's lifetime plus 70 years, or 95 years for works for hire [22].

4.3.2. Open Source Software (OSS)

In contrast to traditional proprietary licenses, used by companies to retain control of their software and restrict access and rights to it outside of the company, OSS licenses are designed to encourage sharing of software and to grant as many rights as possible. OSS licenses may be classified as academic or reciprocal. The academic licenses, including the Berkeley Software Distribution (BSD) license, the Massachusetts Institute of Technology (MIT) license, the Apache Software License (ASL), and the Artistic License, grant nearly all rights and impose few obligations. Anyone can use the software, create derivative works from it, or include it in proprietary projects; typically the obligations are to not remove the copyright and license notices from the software.

Reciprocal licenses encourage sharing of software in a different way by imposing the condition that

the reciprocally-licensed software not be combined (for varying definitions of “combined”) with any software that is not then released in turn under the reciprocal license. The goal is to ensure that as open software is improved, by whomever and for whatever purpose, it remains open. The means is by preventing improvements from vanishing behind closed, proprietary licenses. Examples of reciprocal licenses are GPL, MPL, and CPL.

Licenses of both types typically disclaim liability, assert that no warranty is implied, and obligate licensees to not use the licensor’s name or trademark. Newer licenses tend to discuss patent issues, either giving a limited patent license along with the other rights, or stating that patent rights are not included.

Several newer licenses add interesting degrees of flexibility. Most licenses grant the right to sub-license under the same license, or in some cases under any version of the same license. IBM’s CPL grants the right to sub-license under any license that meets certain conditions; CPL itself meets them, of course, but several other licenses do as well. Finally, the Mozilla Disjunctive Tri-License licenses the core Mozilla components under any one of three licenses (MPL, GPL, or the GNU Lesser General Public License LGPL); OSS developers can choose the one that best suits their needs for a particular project and component.

The Open Source Initiative (OSI) maintains standards for OSS licenses, reviews OSS licenses under those standards, and gives its approval to those that meet them [17]. OSI publishes a standard repository of approximately 70 approved OSS licenses.

It has been common for OSS projects to require that developers contribute their work under conditions that ensure the project can license its products under a specific OSS license. For example, the Apache Contributor License Agreement grants enough of each author’s rights to the Apache Software Foundation for the foundation to license the resulting systems under the Apache Software License. This sort of license configuration, in which the rights to a system’s components are homogenously granted and the system has a well-defined OSS license, was the norm and continues to this day.

4.3.3. Open Architecture (OA)

Open architecture (OA) software is a customization technique introduced by Oreizy [16] that

enables third parties to modify a software system through its exposed architecture, evolving the system by replacing its components. Almost a decade later, we see more and more software-intensive systems developed using an OA strategy not only with open source software (OSS) components but also with proprietary components with open APIs (e.g. [21]). Developing systems using the OA technique can lower development costs [19]. Composing a system with HLS components, however, increases the likelihood of liabilities stemming from incompatible licenses. Thus, in this paper, we define an OA system as a software system consisting of components that are either open source or proprietary with open API, whose overall system rights at a minimum allow its use and redistribution.

OA may simply seem to mean software system architectures incorporating OSS components and open application program interfaces (APIs). But not all such architectures will produce an OA because the available license rights of an OA depend on (a) how and why OSS and open APIs are located within the system architecture, (b) how OSS and open APIs are implemented, embedded, or interconnected, and (c) the degree to which the licenses of different OSS components encumber all or part of a software system's architecture into which they are integrated [1, 19].

The following kinds of software elements appearing in common software architectures can affect whether the resulting systems are open or closed [2].

Software source code components—These can be either (a) standalone programs, (b) libraries, frameworks, or middleware, (c) inter-application script code such as C shell scripts, or (d) intra-application script code, such as for creating Rich Internet Applications using domain-specific languages such as XUL for the Firefox Web browser [6] or "mashups" [15]. Each may have its own license.

Executable components—These components are in binary form, and the source code may not be open for access, review, modification, or possible redistribution [18]. If proprietary, they often cannot be redistributed, and so are present in the design- and run-time architectures but not at distribution-time.

Software services—An appropriate software service can replace a source code or executable component.

Application program interfaces/APIs—Availability of externally visible and accessible APIs is the minimum requirement to form an “open system” [14]. APIs are not and cannot be licensed, and can limit the propagation of license obligations.

Software connectors—Software whose intended purpose is to provide a standard or reusable way of communication through common interfaces, e.g. High Level Architecture [13], CORBA, MS .NET, Enterprise Java Beans, and GNU Lesser General Public License (LGPL) libraries. Connectors can also limit the propagation of license obligations.

Methods of connection—These include linking as part of a configured subsystem, dynamic linking, and client-server connections. Methods of connection affect license obligation propagation, with different methods affecting different licenses.

Configured system or subsystem architectures—These are software systems whose internal architecture may comprise components with different licenses, affecting the overall system license. To minimize license interaction, a configured system or sub-architecture may be surrounded by what we term a license firewall, namely a layer of dynamic links, client-server connections, license shims, or other connectors that block the propagation of reciprocal obligations.

Figure 1 provides an overall view of a reference architecture that includes all of the software elements above. This reference architecture has been instantiated in a number of configured systems that combine OSS and closed source components. One such system handles time sheets and payroll at the university, another implements the web portal for a university research lab (http://proxy.arts.uci.edu/gamelab/).

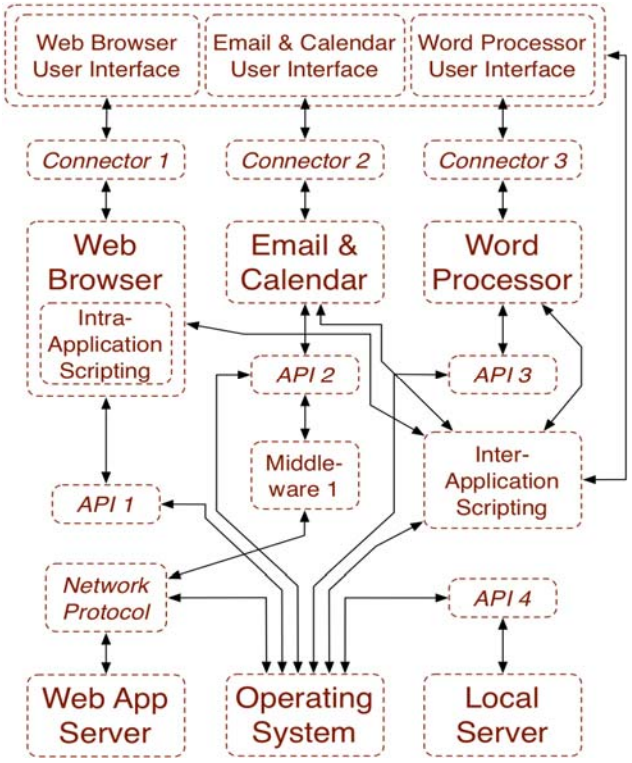


Figure 1. Reference architecture for a heterogeneously-licensed e-business system; connectors (which have no license) are italicized

The configured systems consist of software components such as a Mozilla Web browser, Gnome Evolution email client, and WordPerfect word processor all running on a Linux operating system accessing file, print, and other remote networked servers such as an Apache Web server. The components are interconnected through a set of software connectors that bridge the interfaces of components and combine the provided functionality into the system's services.

4.4. Related work

There has been little explicit guidance on how best to develop, deploy, and sustain complex software systems when different OA and OSS objectives are at hand. Ven [23] and German [8] are recent exceptions.

Ven discusses the challenges faced by independent software vendors who develop software using OSS and proprietary components, focusing on the evolution and maintenance of modified OSS components [23].

German models a license as a set of grants, each of which has a set of conjoined conditions necessary for the grant to be given [8]. Interaction between licenses is analyzed by examining pairs of licenses in the context of five types of component connection. He also identified twelve patterns for avoiding license mismatches, found in a large group of OSS projects, and characterized the patterns using their model. Our license model extends German's to address semantic connections between obligations and rights.

Legal scholars have examined OSS licenses and how they interact in the legal domain, but not how licenses apply to specific HLSs and contexts [7, 20]. For example, Rosen surveys eight existing OSS licenses and creates two more of his own, the Open Source License and the Academic Free License, written to professional legal standards [18]. He examines license interactions primarily in terms of the categories of reciprocal and non-reciprocal licenses, rather than in terms of specific licenses.

Breaux et al. have analyzed regulatory rules in another domain, that of privacy and security [3, 4]. We adapt their approach in our analysis of OSS licenses.

Our previous work examines how best to align acquisition, system requirements, architectures, and OSS elements across different software license regimes to achieve the goal of combining OSS and OA [19].

4.5. Analyzing software licenses

A particularly knotty challenge is the problem of heterogeneous licenses in software systems. In order to illuminate the specifics of this challenge and provide a basis for addressing it, we analyzed a representative group of common OSS licenses and (for contrast) a proprietary license, using an approach based on Breaux's semantic parameterization [4].

We analyzed these licenses:

1. Apache 2.0
2. Berkeley Software Distribution (BSD)
3. Common Public License (CPL)
4. Eclipse Public License 1.0

5. GNU General Public License 2 (GPL)
6. GNU Lesser General Public License 2.1 (LGPL)
7. MIT
8. Mozilla Public License 1.1 (MPL)
9. Open Software License 3.0 (OSL)
10. Corel Transactional License (CTL)

We obtained the text of the nine OSS licenses from the Open Source Initiative web site [17], and the text of the proprietary CTL license from Corel's web site [5].

The stages of the analysis were:

- First we disambiguated forward and backward references, identified synonyms, and distinguished polysemes that expressed different meanings with identical wording. We identified terms of art from copyright law, such as "Derived Work", and specialized terms defined for a particular license, such as "work based on the Program" for GPL and "Electronic Distribution Mechanism" for MPL. From this we constructed (automatically) a concordance to aid us in the remainder of the analysis. The concordance indexed the instances of each distinguished word term, excluding minor words such as articles, conjunctions, and prepositions whose use in a particular license carried no specialized meaning, and tagged each sentence with its section, paragraph, and sentence sequence numbers. Figure 2 shows a portion of the concordance for GPL.

0. S2.0p1s1 This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General_Public_License.
S2.0p1s2 The "Program", below, refers to any such program or work, and a "work_based_on_the_Program" means either the Program or any derivative_work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language.
S2.0p1s3 (Hereinafter, translation is included without limitation in the term "modification".) S2.0p1s4 Each licensee is addressed as "you".

Figure 2. GPL 2 concordance, sect. 2.0 par. 1

- Next we identified the parts of each license that had no legal force, such as GPL 2's "Preamble" section, or the parts that dealt with any rights or obligations other than those for

copyright, such as patents, trademarks, implied warranty, or liability, iterating with the concordance to confirm the identifications. The remainder of our analysis focused on copyright.

- Using the concordances across the licenses, and guided by legal work on OSS licenses [7, 18, 20], we identified words and phrases with the same intentional meaning and textual structures parallel among the licenses. From these we iterated to identify natural language patterns each of which could be used as a restricted natural language statement (RNLS) to express the licenses.

Our meta-model, derived from the patterns we identified, is shown in Figure 3. A license consists of one or more rights, each of which entails zero or more obligations. Rights and obligations have the same structure, a tuple comprising an actor (the licensor or licensee), a modality, an action, an object of the action, and possibly a license referred to by the action.

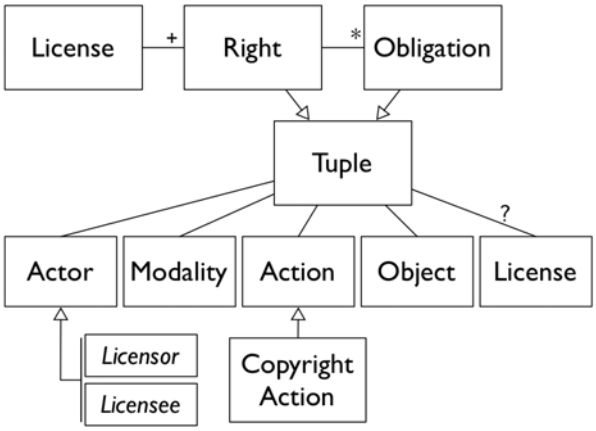


Figure 3. The meta-model for licenses

We found a wide variety of license actions, some of which are defined in copyright law or derived from it and are distinguished as copyright actions. The possible modalities, objects, and licenses are shown in Figure 4.

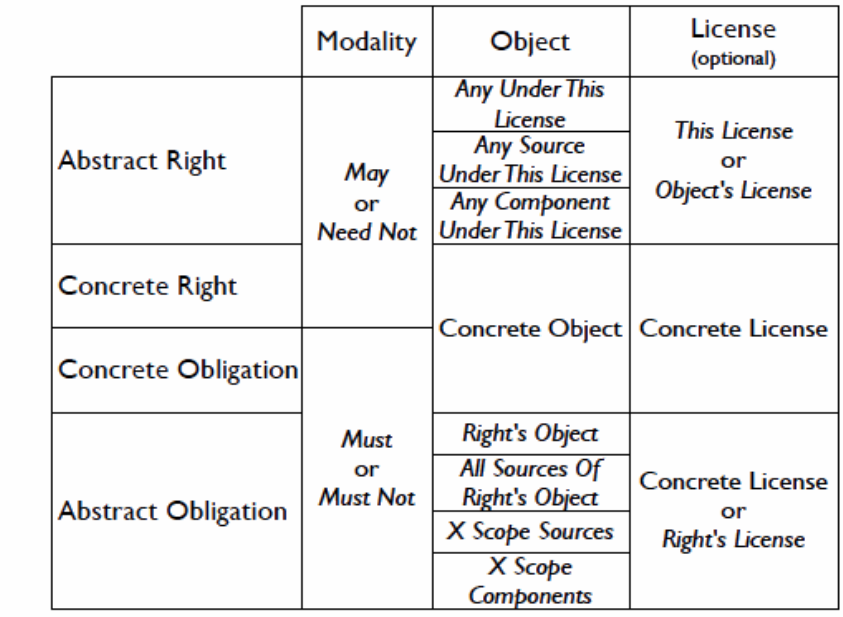


Figure 4: Modality, object, and license

The RNLS textual form of an example abstract right, (one not bound to a specific object) extracted from the BSD license is

Licensee · may · distribute <Any Source> under <This License>

where “distribute under” is a copyright action and the abstract object <Any Source> quantifies the right over all sources licensed under the license containing the right (here, BSD). An example concrete obligation is

Licensee · must · retain the [BSD] copyright notice in [file.c]

where “retain the copyright notice” is an action that is not a copyright action, BSD is the concrete license the action references, and file.c is the concrete object the action references. The RNLS actions are defined with tokens identifying where the tuple’s object and (if present) license are inserted, for example in the GPL action “sub-license % under ö” which becomes “sub-license OBJECT under LICENSE”.

Figure 5 is an informal illustration of how actions may contain concrete objects and licenses,

references to objects or licenses bound elsewhere, or quantifiers using the information in the license architecture abstraction described below to produce sets of rights or obligations.

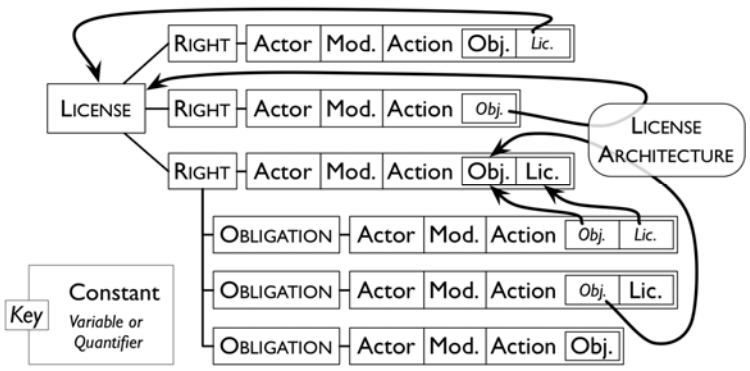


Figure 5. Object/license references, informally

We used the meta-model to express the software licenses and their rights, obligations, and lower level components as Java objects. The constants for the two actors, the four modalities, and the two license quantifiers were implemented as singleton objects of classes that implemented their semantics. Copyright actions became defined constants of the Action class, while the remaining non-copyright actions were unified if their intentional meanings were identical. From this basis we constructed singleton objects for each license, reusing the same object for each instance of its concept in the licenses.

From our analysis we confirmed that the copyright actions form a partial order in which a higher copyright implies the rights it is connected to below it.

Figure 6 shows a portion of the copyright partial order, using brief phrases to identify each defined action. The relation defining the order is “implied by”. For example, “copy” is implied by “sub-license unmodified” because it accomplishes nothing to sub-license copies without making copies, so if we are obligated to sub-license an unmodified component but fail to have the right to copy it, we cannot meet our obligation and do not have whatever rights demand it. The copyright actions are the ones specifically mentioned in the Copyright Act [22]; we incorporated all actions appearing in the licenses we analyzed into the full ordering.

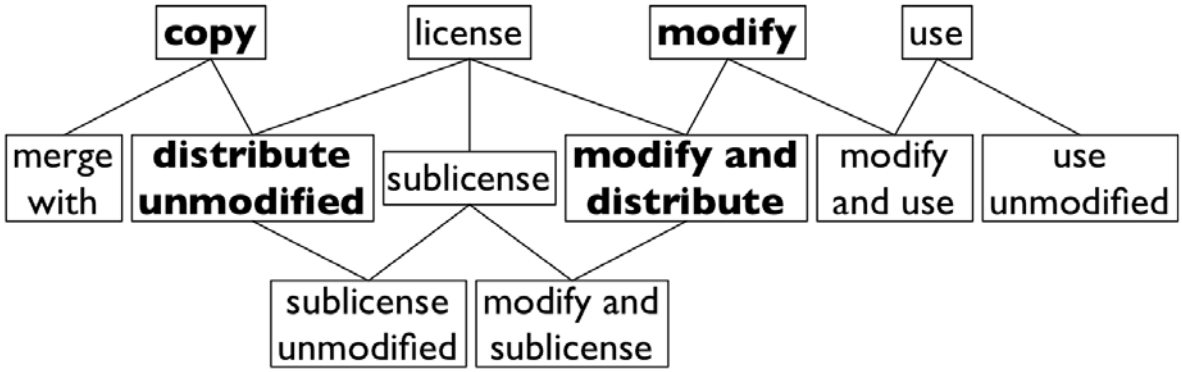


Figure 6. Partial order of copyright actions; actions defined in the Copyright Act in bold

This model of licenses gives a basis for reasoning about licenses, applying them to actual systems, and calculating the results. The additional information we need about the system is defined by the list of quantifiers that can appear as objects in the rights and obligations. The information needed is the license architecture (LA), an abstraction of the system architecture:

1. the set of components of the system,
2. the relation mapping each component to its license,
3. the relation mapping each component to its set of sources, and
4. the relation from each component to the set of components in the same license scope, for each license for which “scope” is defined (e.g. GPL), and from each source to the set of sources of components in the scope of its component (Figure 7).

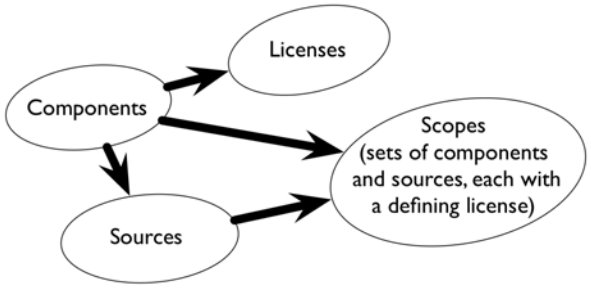


Figure 7. The license architecture meta-model

With this information and definitions of the licenses involved we calculate rights and obligations for individual components or for the entire system and guide HLS design.

We note that the obligation quantifiers we identified in OSS licenses include ones that can set up

conflicts through the license scopes as is well known for GPL and other reciprocal licenses. However, we also identified an obligation quantifier over all sources of a component and note that it raises the possibility of a conflict arising through components that share a source. We believe this conflict path is novel, and are investigating in what contexts, if any, it could occur.

4.6. Analyzing license architectures

In order to make calculations about the rights and obligations for a specific system, we iterate over its components, instantiating each component's license with the component's information. From the resulting concrete rights and obligations, we can determine the set of rights available for the system as a whole and the set of concrete obligations that must be met in order to get those rights.

The instantiation proceeds conceptually as follows.

Each of the abstract rights in every license has as its object either "Any Under This License", "Any Source Under This License", or "Any Component Under This License".

An abstract right R in license L is made into one or more concrete rights by replacing "Any Component" with each component licensed under L in succession, "Any Source" similarly with sources, and "Any" with either. If the abstract right R's license is "Object's License", then in each concrete right r the license is replaced by r's object's license.

Each of R's obligations O is made into one or more concrete obligations o for each r. If O's object is "Right's Object", then there will be a single o, and r's object is used as its object. If O's object is "All Sources Of Right's Object", then there will be an o for each source s of r's object (which must be a component), and that o's object will be s. If O's object is "L' Scope Components" for some license L0, then there will be an o for each component c in r's object's scope under the definition of "scope" in L0, and that o's object will be c, and if O's object is "L' Scope Sources" then analogously for each such component's sources.

However, we do not yet know how to fulfill these concrete obligations. We generate the correlative right from the correlate of the obligation's modality ("may" for "must", "need not" for "must not") and the remaining parts of the obligation. If the correlative right is a copyright right, we must use its object's license to fulfill it: we seek the license of this right's object, find an abstract right that

generalizes this right, and iterate the process for this abstract right and the parts of the correlative right. If there is no such right in the license, we iterate it again for the right's successive containing rights in the partial order of copyright rights, hoping to get all of the rights that include it. If not, we save the correlative right as an unfulfilled correlative right.

If the correlative right is not a copyright right, we do not have to obtain it through a license. However, we must still check whether it conflicts with an obligation. After all the obligations have been determined, we compare it against them, looking for the opposite obligation, the obligation that matches the right's actor, action, object, and modality, but which has the opposite modality ("must not" for "may", "must" for "need not"). If we find such an obligation, then there is a right-obligation conflict.

Finally, we go through the concrete obligations looking for pairs of obligations identical except for their modalities. A pair of such obligations indicates an obligation-obligation conflict.

The presence of any unfulfilled correlative rights, right-obligation conflicts, or obligation-obligation conflicts indicates that the full set of license rights can't be achieved for all of the components in the system. However, we may not need the full set of rights. More commonly we need a subset of the copyright rights, for example the rights to use and distribute the system. This is equivalent to having that set of rights for each component of the system, determined by examining each component's rights for the desired rights.

If the rights are those desired, then the collected concrete obligations for them forms the basis for automated testing that IP obligations have been met [19].

The process outlined above is not one that developers would be keen to follow manually. In the next section we discuss an approach for automating it, both to examine its potential usefulness as an HLS development tool and to validate the analysis process.

4.7. Automating the analysis

We implemented the analysis tool within the traceability view of the ArchStudio software architecture environment [10]. ArchStudio uses an architecture description language (i.e., xADL [11]) to rigorously represent a software system and to enable architecture analysis. We extended

the xADL schema to incorporate the license information. The components are then associated with the license as well as the source. Sub-architectures are used to model scopes. This approach provides:

- The ability to model software systems and specify the corresponding licenses at different levels of granularity. We provide the option of specifying licenses at a fine-grained level, for example licenses assigned to components at the level of a single Web service, such as the Google Desktop Query API, or at a coarse-grained level, for example one license assigned to a set of services provided by Google Desktop APIs, at http://code.google.com/apis/desktop/.
- The ability to model software systems at different architecture levels and to analyze license interactions across the different architecture levels. For instance, if a sub-subsystem X contains heterogeneous licenses and is itself part of a bigger system Y with heterogeneous licenses, our approach is able to analyze license interactions between sub-subsystem X and System Y. We expect to analyze license interactions across multiple architecture levels.
- The modeling approach maps well to the way real software systems are configured.
- Automated license analysis is informed by the additional knowledge of the system configuration. This is one of our contributions beyond current techniques and approaches. Simply modifying the system configuration can result in different sets of available rights or required obligations. Thus, the same set of components may be analyzed with or without specific license firewalls inserted among them.

Figure 8 shows an analysis of a design that specializes the reference architecture in Figure 1.

Scalability is always an issue for any approach. We conclude that our initial algorithm is quadratic in the number of components with licenses, which for architectures of up to several hundred components is manageable. The approach requires modeling the system architecture, in common with many other research approaches, and annotating it to produce the license architecture, which we feel is a worthwhile tradeoff for developers following a best-of-breed strategy or who need to manage reciprocal and proprietary components or design-, distribution-, and run-time architectures that differ in significant ways.

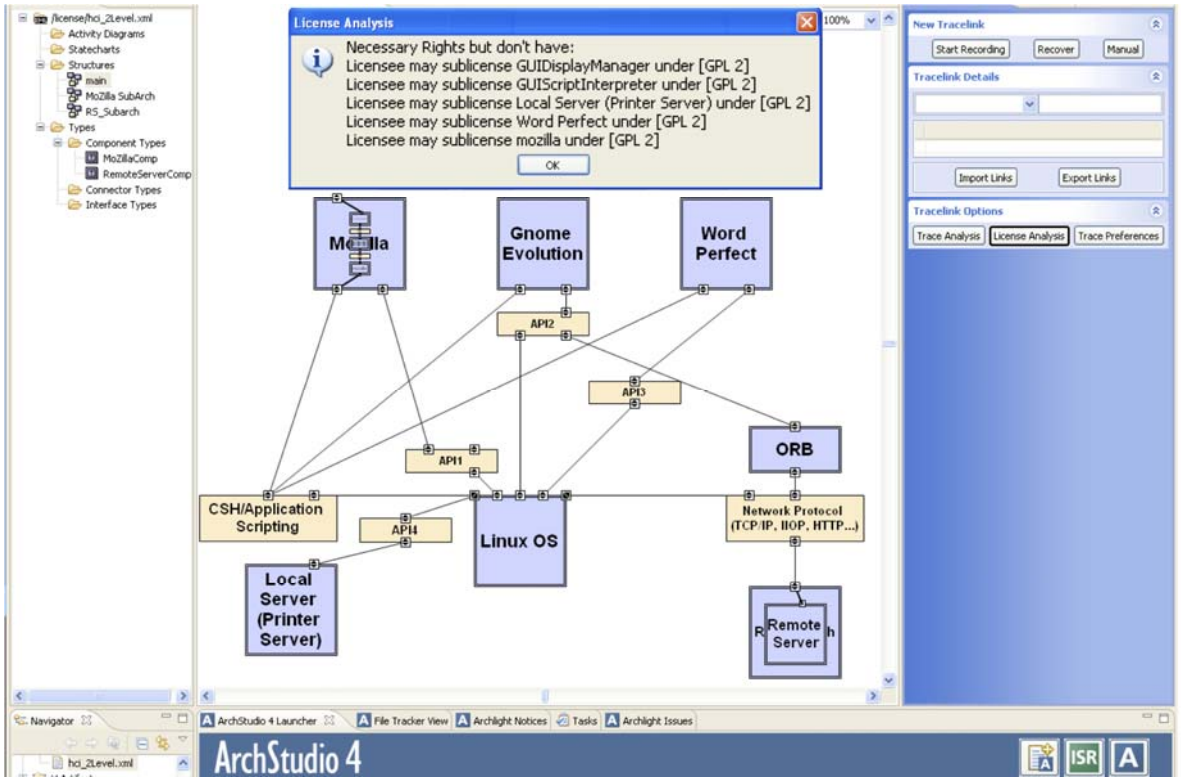


Figure 8. The license architecture analysis tool identifying unavailable rights

The integration of the analysis with architecture design and evaluation supports easy management of licenses across the software development life cycle and across product variations. For instance, as the software evolves, analysts may consider replacing a proprietary word processing component with an OSS component. By simply modifying the architecture model and running the automated license analysis, the analyst learns the new set of rights and obligations. Similarly, an analyst can create product variations to suit a particular deployment platform or customer IP requirements. These product variations can be stored with ArchStudio and retrieved or analyzed at any later time.

4.8. Discussion

Our efforts in this study are motivated in part by a desire to understand how best to accommodate the development of complex software systems whose components may be subject to different IP licenses. These licenses stipulate the rights and obligations that must be ensured. However, system composition can incorporate components with different licenses at architectural design-time, at distribution-time, or at installed release run-time. Thus, we must consider what overall

license schemes we can accommodate, as well as identify the consequences (freedoms and constraints) each scheme can realize.

There are at least two kinds of software license/IP schemes that impose requirements on how software systems will be developed: (a) a single license for the complete software system, and (b) a heterogeneous license scheme of rights and obligations for the complete system incorporating components with different licenses. We consider each in turn.

A single license scheme—There is often a desire to specify a single license at architecture design-time in order to insure a composed software system with single license compatible scheme at distribution-time, and also at run-time. Software licenses like GPL encourage this as part of their overall IP strategy for insuring software freedom. Similarly, there is desire to determine whether a single known license can cover a designed or released system [8]. However, a single license regime cannot in general be guaranteed to occur by chance; instead it is most effectively determined by design. In either case, it must be specified as a nonfunctional requirement for software development. But satisfying such a requirement limits the choice of software components that can be included in the system design and the system composition at distribution-and run-time to those compatible (or subsumed) with the required overall system license. Consequently, our goal in this case is to insure a simple, homogeneous scheme relying on known licenses to determine the propagation and enforcement of their constraints.

A heterogeneous license scheme—In contrast to a single license scheme, a heterogeneous license scheme allows a software system to incorporate components with different IP licenses. Such a scheme gives more degrees of freedom than a single license scheme. For example, it allows for best-of-breed component selection, considering components with a range of licenses rather than only those with a specific license. It also allows for specification and design of software systems conforming to a reference architecture [2]. This enables a higher degree of software reuse through inclusion of reusable software components that have a substantial prior investment in their development and use. Similarly, when relying on a reference architecture, design-time component choices need not be encumbered by license constraints because the resulting system license rights and obligations need only be determined at distribution-time and run-time. Furthermore, the distribution- and run-time system compositions are not limited to a single license, instead they are constrained only by the license rights and obligations that ensue for the entire system.

In a heterogeneous license scheme, the overall system rights and obligations can form a virtual license—a license whose rights and obligations can be determined, tested, and satisfied at any time, without being a previously approved license type (e.g., via the OSI license approval scheme [17]).

This enables prototyping both software system compositions and new software license types, and determining their effect when later mixed with existing software components or licenses. However, determining the scope of rights and obligations in an overall composed system will be challenging without an automated tool such as the one we demonstrated.

The key observation is that there is a choice of ways to proceed in terms of guidance both for those who seek a single license regime for all components and system compositions, as in GPL-based software, and for those who seek to work with multiple software component licenses in order to develop the best possible system designs they can realize.

Finally, it now appears possible to design a pure software IP requirements analysis tool whose purpose is to reconcile the rights and obligations of different licenses, whether new or established. Such a tool will not depend on specific software architectures or distributions for analysis. It may be of value especially to legal scholars or IP lawyers who want to design or analyze alternative IP rights and obligations schemes, as well as to software engineers who want to develop systems with assurable IP rights and obligations. That tool is beyond the scope of our effort here. But it is noteworthy that such a tool can emerge from careful analysis of the requirements for open architecture software systems of HLS components.

4.9. Conclusion

Software licenses and IP rights represent a new class of nonfunctional requirements that increasingly constrain the development of heterogeneously-licensed systems. There has been no model to support analysis and management of these requirements in the context of specific systems, and the heuristics used in practice to deal with them unnecessarily limit the design and implementation choices. It has not been possible to follow a best-of-breed strategy in selecting components without unduly constraining architectural decisions.

In this paper we have presented a meta-model for software licenses through which they can be

made the bases of IP rights calculations on system architectures. We defined the concepts of a license firewall, license architecture, and virtual license, providing a basis for analyzing and understanding the issues involved in HLS IP rights. We outlined an algorithm for calculating concrete rights and obligations for system architectures that identifies conflicts and needed rights, and we validated the meta-model and algorithm by formalizing licenses, incorporating model, licenses, and algorithm into the ArchStudio environment and calculating IP rights and obligations for an existing reference architecture and an instantiation.

Future work includes abstracting our approach to work with licenses in the absence of specific systems, extending it to patent aspects of OSS licenses and applying it to the challenges of software acquisition.

Acknowledgments

This research is supported by grants #0534771 and #0808783 from the U.S. National Science Foundation and grant #N00244-10-1-0038 from the Acquisition Research Program at the Naval Postgraduate School. No review, approval, or endorsement implied.

The authors thank Paul N. Otto for reviewing the discussion of IP law; all remaining faults are due to the authors.

THIS PAGE INTENTIONALLY LEFT BLANK

4.10. References

[1] T. A. Alspaugh and A. I. Antòn. Scenario support for effective requirements. Inf. and Softw. Tech., 50(3):198–220, Feb. 2008.

[2] L. Bass, P. Clements, and R. Kazman. Software Architecture in Practice. Addison-Wesley, 2003.

[3] T. D. Breaux and A. I. Antòn. Analyzing regulatory rules for privacy and security requirements. IEEE Transactions on Software Engineering, 34(1):5–20, 2008.

[4] T. D. Breaux, A. I. Antòn, and J. Doyle. Semantic parameterization: A process for modeling domain descriptions. ACM Trans. on Softw. Eng. and Meth., 18(2), 2008.

[5] Corel Transactional License, 2008. http://apps.corel.com/clp/terms.html .

[6] K. Feldt. Programming Firefox: Building Rich Internet Applications with Xul. O'Reilly Media, Inc., 2007.

[7] R. Fontana, B. M. Kuhn, E. Moglen, M. Norwood, D. B. Ravicher, K. Sandler, J. Vasile, and A. Williamson. A Legal Issues Primer for Open Source and Free Software Projects. Software Freedom Law Center, 2008.

[8] D. M. German and A. E. Hassan. License integration patterns: Dealing with licenses mismatches in component- based development. In 28th International Conference on Software Engineering (ICSE '09), May 2009.

[9] D. Hinchcliffe. Assembling great software: A round-up of eight mashup tools, Sept. 2006.

[10] Institute for Software Research. ArchStudio 4. University of California, Irvine. http://www.isr.uci.edu/projects/archstudio/ .

[11] Institute for Software Research. xADL 2.0. University of California, Irvine. http://www.isr.uci.edu/projects/xarchuci/ .

[12] C. Kanaracus. Adobe readying new mashup tool for business users. InfoWorld, July 2008.

[13] F. Kuhl, R. Weatherly, and J. Dahmann. Creating computer simulation systems: an introduction to the high level architecture. Prentice Hall, 1999.

[14] B. C. Meyers and P. Oberndorf. Managing Software Acquisition: Open Systems and COTS Products. Addison-Wesley, 2001.

[15] L. Nelson and E. F. Churchill. Repurposing: Techniques for reuse and integration of interactive systems. In Int. Conf. on Information Reuse and Integration (IRI-08), page 490, 2006.

[16] P. Oreizy. Open Architecture Software: A Flexible Approach to Decentralized Software Evolution. PhD thesis, University of California, Irvine, 2000.

[17] Open Source Initiative, 2008. http://www.opensource.org/.

[18] L. Rosen. Open Source Licensing: Software Freedom and Intellectual Property Law. Prentice Hall, 2005.

[19] W. Scacchi and T. A. Alspaugh. Emerging issues in the acquisition of open source software within the U.S. Department of Defense. In 5th Annual Acquisition Research Symposium, May 2008.

[20] A. M. St. Laurent. Understanding Open Source and Free Software Licensing. O'Reilly Media, Inc., 2004.

[21] Unity End User License Agreement, Dec. 2008. http://unity3d.com/unity/unity-end-user-license-2.x.html

[22] U.S. Copyright Act, 17 U.S.C., 2008. http://www.copyright.gov/title17/

[23] K. Ven and H. Mannaert. Challenges and strategies in the use of open source software by independent software vendors. Inf. and Softw. Tech., 50(9-10):991–1002, 2008.

5. The Future of Research in Free/Open Source Software Development

Walt Scacchi
Institute for Software Research
University of California, Irvine
Irvine, CA 92697-3455 USA
+1-949-824-4130
wscacchi@ics.uci.edu

Abstract

Free/Open Source Software (FOSS) development is not the same as Software Engineering (SE). Why this is so is unclear and open to various interpretations. Both address the challenges of developing large software systems, but the development processes, work practices, and project forms differ significantly and in interesting ways according to recent empirical studies. This paper reports on highlights from a workshop held in early 2010 on the future of research in FOSS, and how such research relates to or informs our understanding of FOSS and SE, collaborative software development work, software evolution, and new software ecosystems. FOSS and SE are complementary in many ways, yet different in others, so understanding these complements and differences can help advance the future of research in both FOSS and SE. Some of these complements and differences are identified in this paper.

Published as: W. Scacchi, The Future of Research in Free/Open Source Software Development, in *Proc. ACM Workshop on the Future of Software Engineering Research* (FoSER), Santa Fe, NM, 315-319, November 2010.

5.1. Introduction

Even though free/open source software (FOSS) is widely used, much of the Computer Science research community has yet to fully recognize its potential to change the world of research and development of software-intensive systems across disciplines. As little as four years ago, FOSS was still somewhat marginal to SE, appearing in one case as simply another concurrent development methodology rather than as a new approach to software development [1]. However, tens of thousands of FOSS projects are up and running world-wide and millions of end-users of computing increasingly rely on FOSS-based systems. Growing numbers of research projects in physical, social, and human sciences as well as the cultural arts are now routinely expecting to develop or use FOSS-based systems to best meet their needs. Similarly, growing numbers of businesses and government organizations are now looking to develop and use mission-critical software applications that are built with FOSS components. Why and to what ends?

In February 2010, The Computing Community Consortium sponsored a three day workshop to develop an agenda for future research in free/open source software development (FOSSD), in order to better understand the interest, diversification, and widespread growth of FOSS systems, projects, and related practices [2]. The goal of this workshop was to engage researchers from academia and industry in the U.S. to help develop a broad perspective as to emerging areas for potential research investment over the next 5–10 years [13]. The Final Report from this workshop [12] elaborates four core research areas for study as well as the research infrastructure and data needed to support empirical studies of the artifacts, work practices, development processes, project, and community dynamics that characterize FOSSD. The four areas cover FOSSD and Software Engineering (SE), collaborative development work, software evolution, and new software ecosystems. Though it is not possible to cover all of these topics here in appropriate detail, I instead provide a snippet of research results and opportunities for future research at the intersection of FOSSD and SE that are part of the working set of outcomes and research agenda items that were addressed in the workshop and Final Report.

5.2. Sample Research Results on FOSSD

FOSSD is certainly not a panacea for developing complex software systems, nor is it simply SE, or SE done poorly. Instead, it represents an alternative community-intensive socio-technical approach to develop software systems, artifacts, and social relationships. However, it is not without its limitations and

constraints.

First, in terms of participating, joining, and contributing to FOSSD projects, an individual developer's interest, motivation, and commitment to a project and its contributors is dynamic and not indefinite [8]. Some form of reciprocity and self-serving or intrinsic motivation seems necessary to sustain participation, whereas a perception of exploitation by others can quickly dissolve a participant's commitment to further contribute, or worse to dissuade other participants to abandon an open source project that has gone astray.

Second, in terms of cooperation, coordination, and control, FOSSD projects do not escape conflicts in technical decision-making, or in choices of who gets to work on what, or who gets to modify and update what. Because FOSSD projects generally lack traditional project managers, they must become self-reliant in their ability to mitigate and resolve outstanding conflicts and disagreements. Beliefs and values that shape system design choices, as well as choices over which software tools to use and which software artifacts to produce or use, are determined through negotiation rather than administrative assignment. Negotiation and conflict management then become part of the cost that FOSS developers must bear in order for them to have their beliefs and values fulfilled. It is also part of the cost they bear in convincing and negotiating with others, often through electronic communications, to adopt their beliefs and values. Time, effort, and attention spent in negotiation and conflict management represent an investment in building and sustaining a negotiated socio-technical network of dependencies.

Third, forming alliances and building community through participation, artifacts, and tools points to a growing dependence on other FOSSD projects. The emergence of non-profit foundations that were established to protect the property rights of large multi-component FOSSD projects creates a demand to sustain and protect such foundations. If such a foundation becomes too bureaucratic, then this may drive contributors away from its FOSSD projects. So, these foundations need to stay lean, and not become a source of occupational careers, in order to survive and evolve. Similarly, as FOSSD projects give rise to new types of requirements for community building, community software, and community information sharing systems, these requirements need to be addressed and managed by FOSSD project contributors in roles above and beyond those involved in enhancing the source code of a FOSSD project. FOSSD alliances and communities depend on a rich and growing web of socio-technical relations. Thus, if such a web begins to come apart, or if the new requirements cannot be embraced and satisfied, then the FOSSD project community and its alliances will begin to come apart.

Fourth, in terms of the co-evolution of FOSS systems and community, individual and shared resources of people's time, effort, attention, skill, sentiment (beliefs and values), and computing resources are part of the socio-technical web of FOSSD. Reinventing existing software systems as FOSS coincides with the emergence or reinvention of a community who seeks to make such system reinvention occur. FOSS systems are common pool resources that require collective action for their development, mobilization, use, and evolution. Without the collective action of the FOSSD project community, the common pool will dry up, and without the common pool, the community will begin to fragment and disappear, perhaps to search for another pool elsewhere.

Last, empirical studies of FOSSD are expanding the scope of what we can observe, discover, analyze, or learn about how large software systems can be or have been developed. In addition to traditional methods used to investigate FOSSD like reflective practice, industry polls, survey research, and ethnographic studies, comparatively new techniques for mining software repositories [7] and multi-modal modeling and analysis of the socio-technical processes and networks found in sustained FOSSD projects [9, 15] show that the empirical study of FOSSD is growing and expanding. This in turn will contribute to and help advance the empirical science in fields like SE, which previously was limited by restricted access to data characterizing large, proprietary software development projects. Thus, the future of empirical studies of software development practices, processes, and projects will increasingly be cast as studies of FOSSD efforts.

5.3. Research Opportunities for FOSSD and SE

There are a significant number of opportunities and challenges that arise when we look to identifying which software development or socio-technical interaction practices found in studies of FOSSD projects might be applied in the world of SE. As such, let us consider some research opportunities for SE that can arise from FOSSD studies.

First, FOSSD poses the opportunity to favorably alter the costs and constraints of accessing, analyzing, and sharing software process and product data, metrics, and data collection instruments. *FOSSD is poised to fundamentally alter the cost and calculus of empirical SE* [3, 6, 10]. Software process discovery, modeling, and simulation research [e.g., 15] is one arena that can take advantage of such a historically

new opportunity. Similarly, the ability to extract or data mine software product content (source code, development artifacts, team communications, public user feedback) within or across FOSSD project repositories [7] to support its visualization, refactoring, or redesign can be a high-yield, high impact area for SE study and experimentation. Another would be examining the effectiveness and efficiency of traditional face-to-face-to-artifact SE approaches or processes for software inspections [e.g., 16] compared to the online informal peer reviews involving “many eyeballs” prevalent in FOSSD efforts.

Second, based on results from studies of *motivation, participation, role migration, and turnover of individual FOSS developers*, it appears that the SE community would benefit from empirical studies that examine similar conditions and circumstances in conventional software development enterprises. Current SE textbooks and development processes seem to assume that individual developers have technical roles and motivations driven by financial compensation, technical challenge, and the quality assuring rigor that purportedly follows from the use of formal notations and analytical schemes. Said simply, is FOSSD more fun, more interesting, more convivial, and more personally rewarding than SE, and if so, what can be done differently to make SE more like FOSSD?

Third, based on results from studies of *resources and capabilities employed to support FOSSD projects*, it appears that conventional software cost estimation or accounting techniques (e.g., “total cost of operation” or TCO) are limited to analyzing resources or capabilities that are easily quantified or monetized. This in turn suggests that many social and organizational resources/capabilities are slighted or ignored by such techniques, thus producing results that miscalculate the diversity of resources and capabilities that affect the ongoing/total costs of software development projects, whether FOSS or SE based.

Fourth, based on results from studies of *cooperation, coordination, and control practices in FOSSD projects*, it appears that virtual project management and meritocratic socio-technical role migration/advancement can provide a slimmer and lighter weight approach to SE project management. However, it is unclear whether we will see corporate experiments in SE that choose to eschew traditional project management and administrative control regimes in favor of enabling software developers the freedom of choice and expression that may be necessary to help provide the intrinsic motivation to self-organize and self-manage their SE project work.

Fifth, based on results of studies on *alliance formation, inter-project social networking, community*

development, and multi-project software ecosystems, it appears that SE projects currently operate at a disadvantage compared to FOSSD projects. In SE projects, it is commonly assumed that developers and end-users are distinct communities, and that software evolution is governed by market imperatives, the need to extract maximum marginal gains (profit), and resource-limited software maintenance effort. SE efforts are setup to produce systems whose growth and evolution is limited rather than capable of sustaining the exponential growth of co-evolving software functional capability and developer-user community seen in successful FOSSD projects [10].

Last, based on studies of *FOSS as a social movement*, it appears that there is an opportunity and challenge for encouraging the emergence of a social movement that combines the best practices of FOSSD and SE. The world of open source software engineering (OSSE) is the likely locus of collective action that might enable such a movement to arise. For example, the community Web portal for Tigris.org is focused on cultivating and nurturing the emerging OSSE community. More than 700 OSSE projects are currently affiliated with this portal and community. It might therefore prove fruitful to closely examine different samples of OSSE projects at Tigris.org to see which SE tools, techniques, and concepts are being employed, and to what ends, in different FOSSD projects.

5.4. Broader Impact Areas for FOSS Research and Development

There are two major categories of broader impact arising from research in FOSS systems over the next 5–10 years. These are (a) software development and (b) science and industry. Each of these broader impact categories can be described in turn.

5.4.1. Software Development

The development of reliable large, very-large, or ultra-large scale software-intensive systems requires more than robust, formalized, and mathematically grounded approaches to SE. They also require the engagement of decentralized communities of practitioners who can participate in and contribute to the ongoing development, use, and evolution of software system tools, online artifacts, and other information infrastructure resources, either on a local or global basis. The development of software-intensive systems at large-scale and beyond needs to be recognized as something now essential to the advancement of science, technology, industry, government, and society across geographic borders and cultural boundaries.

FOSS systems research is likely to change how SE research and practice are now accomplished. The openness of FOSS system development means that new participants are coming into the world of software systems to contribute to the ongoing development and evolution of such systems. The engagement and contribution of FOSS system development participants who are not necessarily skilled in the traditional principles and practices of SE means there will be a long-term need to adapt SE concepts, techniques, and tools to people lacking skills in SE, while also seeking new ways and means for motivating these new participants to engage in learning and practicing emerging SE processes, practices, and principles. In addition, the public availability of FOSSD artifacts will likely become a primarily source of data for empirical SE research because such data will often be far less encumbered by corporate non-disclosure agreements that have historically limited what software development data can be made available for scientific research purposes.

FOSS systems research will continue to be a rich source of observation and experimentation for collaborative software development processes, practices, and project forms. Because many successful, ongoing, and large-scale FOSS systems and project communities are typically physically decentralized but logically centralized, this ways and means for such sustained software development must rely on collaboration tools, techniques, and patterns of use whose fundamental principles we do not yet fully understand. Yet, FOSS system development is a clear, recurring demonstration that the development of complex systems can be performed, governed, and sustained in a decentralized manner, with little/no corporate oversight or enterprise governance, when corporate oversight and governance regimes have long been a hallmark for the development and maintenance of large complex systems. Collaborative FOSS system development processes, practices, project forms, project infrastructures, and surrounding ecosystem represent new ways and means for developing complex systems that help meet societal needs.

FOSS systems depend on and co-evolve with their surrounding ecosystem. FOSS systems are both social and technological endeavors, where socio-technical interactions are more critical to system development, use, and evolution than a formal mathematical basis for specifying the system's analytical intent. Yet understanding how FOSS system ecosystems operate is at a very early stage of study, and how best to study, explain, and rationalize it is still in a very formative stage. But human-made complex systems are increasingly recognized as being products of their own complex ecosystems and of the networks of producers, integrators, and consumers who create, assemble, and use such systems.

Thus, research into complex system ecosystems like those that situate and embed FOSS systems are within the grasp of scientific study, comprehension, and explanation. These eventual accomplishments would provide the basis for rationalizing, predicting, controlling, and transferring such knowledge to other complex ecosystems, especially those that are mediated by information infrastructures or cyberinfrastructure. Thus, research into FOSS ecosystems is critical to advancing scientific knowledge and technology development in many areas beyond software systems.

Last, FOSS systems are complex software systems with an open evolutionary history and future. Such openness is in many ways historically unprecedented for complex technical systems. So we should not miss or scuttle such a rare opportunity to study FOSS system and ecosystem evolution as a software system, as a decentralized social system for peer production, and as a complex socio-technical system.

5.4.2. Science and Industry

Many grand challenges for science and engineering going forward depend on the research and development of a new generation of complex, software-intensive systems (cf. http://www.engineeringchallenges.org/, accessed September 2010). Advanced healthcare informatics, advanced personalized learning systems, secure cyberspace, engineering automated tools for scientific discovery, and enhanced virtual reality are all readily recognized as problem domains that depend on future software systems. Making solar energy economical, managing the nitrogen cycle, preventing nuclear terror, providing energy from fusion, providing access to clean water, engineering better medicines, developing carbon sequestration methods, improving urban infrastructure, and reverse engineering the human brain are also areas where new generations of software systems are needed to enable and deploy the sought after scientific advances. But meeting these grand challenges depends on more than robust or well-engineered software systems.

It appears likely that the development of software systems in these other application domains will increasingly depend in part or in full on FOSS systems and ecosystems, as well as on FOSSD processes, practices, and project forms. The reasons for this are many, but not inevitable. Social choices and economic constraints may make proprietary or closed source system solutions less practical and less desirable. For example, if scientific research into fusion energy centers around the International Thermonuclear Energy Research (ITER) project, with its forecast budget of more than $100B (and it is still in the early stages of development), then how much of that budget will be allocated

to development of ITER control system software, and who will be called upon to develop or engineer the requisite software? ITER is a multi-national effort and there is likely to be a common call for openness in its software development projects, as well as openness in science practices, rather than an expectation that some company or contractor will be called upon to develop a proprietary, closed source software system. As such, it may be the case that grand challenge problems are more likely to embrace or demand openness in their system development efforts in part or in full, or at least prior to any commercialization of supporting software systems.

FOSS system development has already begun to transform the global software industry and all major software and Information Technology (IT) firms. Proprietary, closed source systems are not likely to disappear, but there will be growing pressure to expect that proprietary systems offer innovative features or functions that are not yet available as FOSS systems. FOSS systems may help to drive technological advances in proprietary systems and closed source system developers out of self-interest and preservation of commercial or market position. Once again, FOSS systems are creative drivers that help stimulate advances to the broader economy and IT marketplace. Companies and firms that actively resist the progressive transition to FOSS systems in different application or service areas will be increasingly marginalized rather than embraced. FOSS systems will increasingly take over mundane, infrastructural, and non-competitive IT domains and this will help to clarify where IT or software system value truly is to be found. So, stimulating research and development into FOSS systems and FOSSD projects are a strategic national investment if the goal is to improve national and industrial IT system capabilities and related industries.

Finally, advances in enterprise information systems that help realize new ways and means for improving or streamlining enterprise operations, creating new products or services, and creating more stimulating jobs, careers, and workforce development opportunities depend on faster, better, and cheaper software systems. Helping to make regional and national government agencies more transparent, open, and trustworthy requires public access to information systems that are easy to access, open for study, and open to citizen participation. FOSS systems are the most likely technology that can realize these societal needs.

5.5. Conclusions

Free and open source software development is emerging as an alternative approach for how to develop

large software systems. FOSSD employs socio-technical work practices, development processes, and community networking often different from those found in industrial software projects and from those portrayed in software engineering textbooks [17]. As a result, FOSSD offers new types and new kinds of practices, processes, and organizational forms to discover, observe, analyze, model, and simulate. Similarly, understanding how FOSSD practices, processes, and projects are similar to or different from traditional SE counterparts is an area ripe for further research and comparative study. Many new research opportunities exist in the empirical examination, modeling, and simulation of FOSSD activities, efforts, and communities.

FOSSD project source code, artifacts, and online repositories represent and offer new publicly available data sources of a size, diversity, and complexity not previously available for SE research on a global basis. For example, software process modeling and simulation research and application has traditionally relied on an empirical basis in real-world processes for analysis and validation. However, such data has often been scarce, costly to acquire, and is often not available for sharing or independent re-analysis for reasons including confidentiality or non-disclosure agreements. FOSSD projects and project artifact repositories contain process data and product artifacts that can be collected, analyzed, shared, and that can be re-analyzed in a free and open source manner.

Last, through surveys of empirical studies of FOSSD projects [4, 5, 11], it should be clear that there are an exciting variety and diversity of opportunities for new research into software development processes, work practices, project/community dynamics, and related socio-technical interaction networks. Thus, you are encouraged to consider how your efforts to engage in SE research or apply FOSSD concepts, techniques, or tools can be advanced through studies that examine FOSSD activities, artifacts, and projects.

Acknowledgments

The research described in this paper has been supported by grant #0808783 from the U.S. National Science Foundation, grant #N00244-10-1-0038 from the Acquisition Research Program at the Naval Postgraduate School, and a grant from the Computing Community Consortium. No review, approval, or endorsement implied.

THIS PAGE INTENTIONALLY LEFT BLANK

5.6. References

[1] Boehm, B.E., A View of 20th and 21st Century Software Engineering. *Proc. 28th International Conference on Software Engineering (ICSE 2006)*, Shanghai, China, 12–29, ACM Press, 2006.

[2] Computing Community Consortium Workshop on Free/Open Source Software Development, http://cra.org/ccc/foss.php and http://foss2010.isr.uci.edu/, 10–12 February 2010, accessed June 2010.

[3] Cook, J.E., Votta, L.G., and Wolf, A.L., Cost-Effective Analysis of In-Place Software Processes, *IEEE Trans. Software Engineering*, 24(8), 650–663, 1998.

[4] Crowston, K., Wei, K., Howison, J., and Wiggins, A. (2010). Free/libre open source software development: what we know and what we do not know. *ACM Computing Surveys*, (in press).

[5] Hauge, O., Ayala, C. and Conradi, R. (2010). Adoption of Open Source Software in Software-Intensive Organizations - A Systematic Literature Review. *Information and Software Technology,* (in press).

[6] Harrison, W., Editorial: Open Source and Empirical Software Engineering, *Empirical Software Engineering*, 6(2), 193–194, 2001.

[7] Howison, J., Conklin, M., and Crowston, K., FLOSSmole: A Collaborative Repository for FLOSS Research Data and Analyses. *Intern. J. Info. Tech. and Web Engineering*, 1(3), 17–26, 2006.

[8] Robles, G. and Gonzalez-Baharona, J.M., Contributor Turnover in Libre Software Projects, in Damiani, E., Fitzgerald, B., Scacchi, W., Scotto, M. and Succi, G., (Eds.), *Open Source Systems*, IFIP Vol. 203, Springer, Boston, 273–286, 2006.

[9] Sack, W., Detienne, F., Ducheneaut, Burkhardt, Mahendran, D., and Barcellini, F., A Methodological Framework for Socio-Cognitive Analyses of Collaborative Design of Open Source Software, *Computer Supported Cooperative Work*, 15(2/3), 229–250, 2006.

[10] Scacchi, W., Understanding Free/Open Source Software Evolution, in N.H. Madhavji, J.F. Ramil and D. Perry (Eds.), *Software Evolution and Feedback: Theory and Practice*, John Wiley and Sons Inc, New York, 181–206, 2006.

[11] Scacchi, W. Free/Open Source Software Development: Recent Research Results and Methods, in M. Zelkowitz (Ed.), *Advances in Computers*, 69, 243–295, 2007.

[12] Scacchi, W., Crowston, K. Jensen, C., Madey, G., Squire, M., et al., *Towards a Science of Open Source Systems*, Final Report, Institute for Software Research, University of California, Irvine, Fall

2010. http://foss2010.isr.uci.edu/content/foss-2010-reports accessed September 2010.

[13] Scacchi, W., Crowston, K., Madey, G., and Squire, M. Envisioning National and International Research on the Multi-Disciplinary Study of Free/Open Source Software, Spring 2009. http://www.ics.uci.edu/~wscacchi/ProjectReports/CCC-FOSS-SPRING2009.pdf, accessed June 2010.

[14] Scacchi, W., Feller, J. Fitzgerald, B., Hissam, S., and K. Lahkani, Understanding Free/Open Source Software Development Processes, *Software Process--Improvement and Practice*, 11(2), 95–105, March/April 2006.

[15] Scacchi, W., Jensen, C., Noll. J. and Elliott, M.E. Multi-Modal Modeling, Analysis and Validation of Open Source Software Development Processes, *Intern. J. Internet Technology and Web Engineering*, 1(3), 49–63, 2006.

[16] Seaman, C.B. and Basili, V., Communication and Organization: An Empirical Study of Discussion in Inspection Meetings, *IEEE Trans. Software Engineering*, 24(6), 559–572, 1998.

[17] Sommerville, I., *Software Engineering, 8th Edition*, Addison-Wesley, New York, 2006.

6. Free/Open Source Software Development

Walt Scacchi
Institute for Software Researcher
University of California, Irvine
Irvine, CA 92697-3455 USA

Abstract

This article examines and reviews what is known so far about free/open source software development (FOSSD). FOSSD is not the same as software engineering that is portrayed in common textbooks. Instead, it is a complementary approach to address the many challenges that arise in the development of complex software systems that are often built outside of a traditional corporate software development environment. This article highlights some of the basic understandings for how FOSSD works based on empirical studies of FOSSD projects, processes, and work practices in different communities. This includes identification of different types of informal online artifacts that facilitate and constrain FOSSD projects. This article also identifies what different studies examine as well as the approaches used to sample and systematically study FOSSD. Next, opportunities for constructive crossover studies of software engineering and FOSSD help reveal new directions for further research study. Finally, the last section presents limitations and conclusions regarding studies of FOSSD.

Published as: W. Scacchi, Open Source Software, *Encyclopedia of Software Engineering*, Taylor and Francis, New York, 614–626, 2010.

6.1. Introduction

This articles examines practices, patterns, and processes that have been observed in studies of free/open source software development (FOSSD) projects. FOSSD is a way for building, deploying, and sustaining large software systems on a global basis and differs in many interesting ways from the principles and practices traditionally advocated for software engineering [26]. Thousands of FLOSS systems are now in use by thousands to millions of end-users, and some of these FLOSS systems entail hundreds of thousands to millions of lines of source code. So, what's going on here, and how is FOSSD being used to build and sustain these projects differently, and how might differences be employed to explain what's going on with FOSSD, and why?

One of the more significant features of FOSSD is the formation and enactment of complex software development processes and practices performed by loosely coordinated software developers and contributors [24]. These people may volunteer their time and skill to such effort and may only work at their personal discretion rather than as assigned and scheduled. However, increasingly, software developers are being assigned as part of their job to develop or support FLOSS systems, and thus to become involved with FOSSD efforts. Furthermore, FLOSS developers are generally expected (or prefer) to provide their own computing resources (e.g., laptop computers on the go, or desktop computers at home) and to bring their own software development tools with them. Similarly, FLOSS developers work on software projects that do not typically have a corporate owner or management staff to organize, direct, monitor, and improve the software development processes being put into practice on such projects. But how are successful FOSSD projects and processes possible without regularly employed and scheduled software development staff, or without an explicit regime for software engineering project scheduling, budgeting, staffing, or management [24]? What motivates software developers to participate in FOSSD projects? Why and how are large FOSSD projects sustained? How are large FOSSD projects coordinated, controlled, or managed without a traditional project management team? Why and how might the answers to these questions change over time? These are the kinds of questions that will begin to be addressed in this article.

The remainder of this article is organized as follows. The next section provides a brief background on what FOSS is and how free software and open source software development efforts are similar and different. From there, attention shifts to identify different types of informal online artifacts that facilitate

and constrain FOSSD projects. The article also identifies what different studies examine as well as the approaches used to sample and systematically study FOSSD. Next, opportunities for constructive crossover studies of software engineering and FOSSD help reveal new directions for further research study. Finally, the article concludes with a discussion of limitations and constraints in the FOSSD studies so far regarding what is known about FOSSD through systematic empirical study.

6.2. *What is free/open source software development?*

Free (as in freedom) software and open source software are often treated as the same thing [8]. However, there are differences between them with regards to the licenses assigned to the respective software. Free software generally appears licensed with the GNU General Public License (GPL), whereas OSS may use either the GPL or some other license that allows for the integration of software that may not be free software. Free software is a social movement, whereas FOSSD is a software development methodology according to free software advocates like Richard Stallman and the Free Software Foundation [10]. Yet some analysts also see OSS as a social movement distinguishable from, but surrounding, the free software movement [6], as in a set to sub-set relationship. The hallmark of free software and most OSS is that the source code is available for public access, open to study and modification, and available for redistribution to others with few constraints, except the right to ensure these freedoms. OSS sometimes adds or removes similar freedoms or copyright privileges depending on which OSS copyright and end-user license agreement is associated with a particular OSS code base. More simply, free software is always available as OSS, but OSS is not always free software. This is why it often is appropriate to refer to FOSS or FLOSS (L for *Libre*, where the alternative term “libre software” has popularity in some parts of the world) in order to accommodate two similar or often indistinguishable approaches to software development. Thus, at times it may be appropriate to distinguish conditions or events that are generally associated or specific to either free software development or OSSD, but not both. Subsequently, for the purposes of this article, focus is directed at FOSSD practices, processes, and dynamics, rather than at software licenses, though such licenses may impinge on them. However, when appropriate, particular studies examined in this review may be framed in terms specific to either free software or OSS when such differentiation is warranted.

FOSSD is mostly not about software engineering (SE), at least not as SE is portrayed in modern SE textbooks [e.g., 26]. FOSSD is also not SE done poorly. It is instead a different approach to the development of software systems where much of the development activity is openly visible and where

development artifacts are publicly available over the Web. Furthermore, substantial FOSSD effort is directed at enabling and facilitating social interaction among developers (and sometimes also end-users), but generally there is no traditional software engineering project management regime, budget or schedule. FOSSD is also oriented towards the joint development of an ongoing community of developers and users concomitant with the FLOSS system of interest.

FLOSS developers are typically also end-users of the FLOSS they develop, and other end-users often participate in and contribute to FOSSD efforts. There is also widespread recognition that FOSSD projects can produce high quality and sustainable software systems that can be used by thousands to millions of end-users [18]. Thus, it is reasonable to assume that FOSSD processes are not necessarily of the same type, kind, or form found in modern SE projects [cf. 24, 26]. Whereas such approaches might be used within an SE project, there is no basis found in the principles of SE laid out in textbooks that would suggest SE projects typically adopt or should practice FOSSD methods. Subsequently, what is known about SE processes, or modeling and simulating SE processes, may not be equally applicable to FOSSD processes without some explicit rationale or empirical justification. Thus, it is appropriate to review what is known so far about FOSSD.

6.2.1. The Early Days of Software Source Code Sharing

The sharing, modification, and redistribution of open software source code did not begin with the appearance of the first Linux software in the early 1990's, nor with the appearance of the Emacs text editor or Gnu C Compiler, GCC (later to evolve into the Gnu Compiler Collection, but still called GCC) by Richard Stallman in the mid 1980's. Instead, the sharing of open software source code dates back to the early days of computing in the 1950's. IBM established *SHARE* as a community of software developers and users of its early mainframe computers who were aided in the efforts to create, share, enhance, reuse, and distribute open software source code [1]. IBM supported SHARE and open software source code until it decided that unbundling software purchases from its mainframe computer sales efforts was in its best interest. The early days of operating systems like *TENEX* for the DecSystem-10 mainframe and later for *Unix* on the DEC PDP-11 series mini-computers in the 1970's also benefited from access, study, modification, and sharing of modifications made by user-developers primarily in the computer science research community. Similarly, efforts to share, reuse, study, enhance, and redistribute then new programming languages, interpreters, or compilers for languages like *Lisp*, *Pascal*, and *C* again with the academic research community in the 1970–1980's was

common because no computer manufacturers showed interest in developing or sustaining such software until their commercial viability as products could be substantiated. Even complete relational database management systems like *Ingres* from UC Berkeley first appeared as open software source code that was available in the public domain in this same time period. Similarly, scientific computing routines were widely shared and reused, and their source code was published in books [20]. As such, the tradition of publishing open application software source code for research or play purposes had already been established and in practice in the 1960's, such a Spencer's *Game Playing With Computers* [27] and the accessible through Internet-based online repositories with the computer-aided design package *BRL-CAD* in 1983. Finally, large, integrated software development environments such as the *USC System Factory Project* incorporated hypertext facilities for navigating software sources and for organizing and storing hypertext-linked software artifacts in team-based, multi-project repositories, also were developed for open access, sharing, study, enhancement, and redistribution over the Internet using minimally restrictive academic software licenses [5,21]. So, the practice of sharing open source software source code that was accessible, available for study and modification, and with some but not universal accommodations for redistributing modified versions the source were all established practices by the early-mid 1980's. But, the mid-1980's to the early 1990's is the period most often identified as the origins of the FOSS, the Free Software Movement, as documented in many forms, including the documentary film, *Revolution OS* [19]. Subsequently, by the mid-late 1990's we began to see the first “revolutionary” writings and studies about the practice of FOSSD that utilize the World-wide Web, the GPL software license, and free software as a social movement [3], though nearly all of it is somehow unaware, forgetful, or ignorant of the open software efforts preceding it.

6.2.2. The Rise of Empirical Studies of FOSSD

The presence of the so-called revolution in software development that FOSS was said to represent challenged many scholars to think about whether what was known about software development practices was still accurate, or whether something new was at hand. These concerns then gave rise to a burst of new studies of FOSSD projects and practices to help determine what the revolution was all about.

Studies and recommended practices for FOSSD often focus on the interests, motivations, perceptions, and experiences of developers or end-user organizations. Typically, attention is directed to the *individual agent* (most commonly a person, unitary group, firm, or some other stakeholder, but

sometimes a software system, tool, or artifact type) acting within a larger actor group or community. Individual behavior, personal choices, or insider views might best be analyzed, categorized, and explained in terms of volunteered or elicited statements of their interests, motivations, perceptions, or experiences. Popular treatments of OSS development [3] and Free software development [9,10], provide insight and guidance for how FLOSS development occurs based on the first-hand experiences of those authors.

Other authors informed by such practitioner studies and informal industry/government polls (like those reported in *CIO Magazine,* the Open Source Business Conference, the Open Solutions Alliance, and elsewhere) seek to condense and package the experience and wisdom of these FLOSS practices into practical advice that may be offered to business executives considering when and why to adopt FLOSS options [13].

Many of these studies and insights about how FOSSD works resulted from reflective practice rather than from systematic empirical study. However, personal reflection on software development practice often tends to (a) be uncritical with respect to prior scholarship or theoretical interpretation or (b) employ the unsystematic collection of data to substantiate pithy anecdotes or proffered conclusions. Thus, by themselves such studies offer a limited basis for further research or comparative analysis. Nonetheless, they can (and often do) offer keen insights and experience reports that may help sensitize future FOSSD researchers to interesting starting points or problems to further explore.

6.3. Understanding FLOSS development across different communities

It is best to assume that there is no general model or globally accepted framework that defines how FLOSS is or should be developed. Subsequently, our starting point is to investigate FLOSS practices in different communities from an ethnographically informed, qualitative perspective [4, 25]. Here we briefly examine four different FOSSD communities to describe and compare. These are those centered on the development of software for networked computer games, Internet/Web infrastructure, scientific computing, and academic software engineering research. The following sections briefly introduce and characterize these FLOSS sub-domains. Along the way, example software systems or projects are *highlighted* or identified via external reference/citation, which can be consulted for further information or review.

6.3.1. Networked computer game worlds

Participants in this community focus on the development and evolution of first person shooter (FPS) games (e.g., *Quake Arena*, *Unreal Tournament*), massive multiplayer online role-playing games (e.g., *World of Warcraft, Lineage, EveOnline, City of Heroes*), and others (e.g., *The Sims* [Electronic Arts], *Grand Theft Auto* [Rockstar Games]). Interest in networked computer games and gaming environments, as well as in their single-user counterparts, has exploded in recent years as a major mode of entertainment, playful fun, popular culture, and global computerization movement. The release of *DOOM*, an early first-person action game, onto the Web in open source form in the mid 1990's, began what is widely recognized as the landmark event that launched the development and redistribution of computer game *mods*—open source and distributable modifications of commercially available games created with software development kits provided with the retail game package by the game's software developer. The end-user license agreement for games that allow for end-user created game mods often stipulate that the core game engine (or retail game software product) is protected as closed source, proprietary software that cannot be examined or redistributed, whereas any user created mod can only be redistributed as open source software that cannot be declared proprietary or sold outright, and that must only be distributed in a manner where the retail game product must be owned by any end-user of a game mod. This has the effect of enabling a secondary market for retail game purchases by end-users or other game modders who are primarily interested in accessing, studying, playing, further modifying, and redistributing a game mod.

Mods are variants of proprietary (closed source) computer game engines that provide extension mechanisms like (domain-specific) game scripting languages (e.g., *UnrealScript* for mod development with *Unreal* game engines from Epic Games Inc.) that can be used to modify and extend a game. Extensions to a game created with these mechanisms are published for sharing across the Web with other game players and are licensed for such distribution in an open source manner. Mods are created by small numbers of users who want and are able to modify games (they possess some software development skills) compared to the huge numbers of players that enthusiastically use the games as provided. The scope of mods has expanded to now include new game types, game character models and skins (surface textures), levels (game play arenas or virtual worlds), and artificially intelligent game bots (in-game opponents). For additional background on computer game mods, see http://en.wikipedia.org/wiki/Mod_(computer_gaming).

Perhaps the most widely known and successful game mod is *Counter-Strike* (CS), which is a total conversion of Valve Software's *Half-Life* computer game. Valve Software has since commercialized *CS* and many follow-on versions. *CS* was created by two game modders who were reasonably accomplished students of software development. Millions of copies of *CS* have subsequently been distributed, and millions of people have played *CS* over the Internet according to http://counterstrikesource.net/. Other popular computer games that are frequent targets for modding include the *Quake, Unreal, Half-Life,* and *Crysis* game engines, *NeverWinter Nights* for role-playing games, motor racing simulation games (e.g., *GTR* series), and even the massively popular *World of Warcraft* (which only allows for modification of end-user interfaces, and not the game itself). Thousands of game mods are distributed through game mod portals like http://www.MODDB.com. In contrast, large successful game software companies like Electronic Arts and Microsoft do not embrace nor encourage game modding, and do not provide end-user license agreements that allow game modding and redistribution.

6.3.2. Internet/Web infrastructure

The SourceForge web portal (http://www.sourceforge.net), the largest associated with the FLOSS community, currently stores information on more than 1,750K registered users and developers, along with nearly 200K FOSSD projects (as of July 2008), with more than 10% of those projects indicating the availability of a mature, released, and actively supported software system. However, some of the most popular FLOSS projects have their own family of related projects, grouped within their own portals, such as for the Apache Foundation and Mozilla Foundation. Participants in this community focus on the development and evolution of systems like the *Apache* web server, *Mozilla/Firefox* Web browser, *GNOME* and *K Development Environment* (KDE) for end-user interfaces, the *Eclipse* and *NetBeans* interactive development environments for Java-based Web applications, and thousands of others. This community can be viewed as the one most typically considered in popular accounts of FLOSS projects. However, it is reasonable to note that the two main software systems that enabled the World-wide Web, the *NCSA Mosaic* Web browser (and its descendants, like Netscape Navigator, Mozilla, Firefox, and variants like *K-Meleon*, *Konqueror*, *SeaMonkey*, and others), and the Apache Web server (originally know as *httpd*) were originally, and still active, FOSSD projects.

The GNU/Linux operating system environment is one of the largest, most complex, and most diverse sub-communities within this arena, so much so that it merits separate treatment and examination.

Many other Internet or Web infrastructure projects constitute recognizable communities or sub-communities of practice. The software systems that are the focus generally are not standalone end-user applications, but are often directed toward *system administrators* or *software developers as the targeted user base,* rather than the eventual end-users of the resulting systems. However, notable exceptions like Web browsers, news readers, instant messaging, and graphic image manipulation programs are growing in number within the end-user community.

6.3.3. Scientific computing in X-ray astronomy and deep space imaging

Participants in this community focus on the development and evolution of software systems supporting the Chandra X-Ray Observatory, the European Space Agency's XMM-Newton Observatory, the Sloan Digital Sky Survey, and others. These are three highly visible astrophysics research projects whose scientific discoveries depend on processing remotely sensed data through a complex network of open source software applications that process remotely sensed data. In contrast to the preceding two development oriented FOSSD communities, open software plays a significant role in scientific research communities. For example, when scientific findings or discoveries resulting from remotely sensed observations are reported, then members of the relevant scientific community want to be assured that the results are not the byproduct of some questionable software calculation or opaque processing trick. In scientific fields like astrophysics that critically depend on software, open source is considered an essential precondition for research to proceed and for scientific findings to be trusted and open to independent review and validation. Furthermore, as discoveries in the physics of deep space are made, this in turn often leads to modification, extension, and new versions of the astronomical software in use that enable astrophysical scientists to further explore and analyze newly observed phenomena or to modify/add capabilities to how the remote sensing or astrophysical computation mechanisms operate. For example, the NEMO Stellar Dynamics Package at http://bima.astro.umd.edu/nemo/ is now at version 3.2.5, as of September 2008.

To help understand these matters, consider the example at http://antwrp.gsfc.nasa.gov/apod/ap010725.html . This Website page displays a composite image constructed from both X-ray (Chandra Observatory) and optical (Hubble Space Telescope) sensors. The open software processing pipelines for each sensor are mostly distinct and are maintained by different organizations. However, their outputs must be integrated and the images must be registered and oriented for synchronized overlay, pseudo-colored, and then composed into a final image as

shown on the cited Web page. Subsequently, there are dozens of open software programs that must be brought into alignment for such an image to be produced and for such a scientific discovery to be claimed and substantiated.

6.3.4. Academic software systems design

Participants in this community focus on the development and evolution of software architecture and UML centered design efforts, such as for ArgoUML (http://argouml.tigris.org) or xARCH at CMU and UCI (http://www.isr.uci.edu/projects/xarch/). This community can easily be associated with a mainstream of SE research. People who participate in this community generally develop software for academic research or teaching purposes in order to explore topics like software design, software architecture, software design modeling notations, software design recovery (reverse software engineering), etc. Accordingly, it may not be unreasonable to expect that open software developed in this community should embody or demonstrate principles from modern SE theory or practice. Furthermore, much like the X-ray astronomy community, members of this community expect that when breakthrough technologies or innovations have been declared, such as in a refereed conference paper or publication in a scholarly journal, the opportunity exists for other community members to be able to access, review, or try out the software to assess and demonstrate its capabilities. An inability to provide such access may result in the software being labeled as “vaporware” and the innovation claim challenged, discounted, or rebuked. Alternatively, declarations of “non-disclosure” or “proprietary intellectual property” are generally not made for academic software unless or until it is transferred to a commercial firm. However, it is often acceptable to find that academic software constitutes nothing more than a “proof of concept” demonstration or prototype system not intended for routine or production use by end-users.

6.3.5. Overall characteristics across different FOSSD communities

In contrast to efforts that draw attention to generally one (but sometimes many) open source development project(s) within a single community [e.g., 4, 7, 12, 18, 23], there is something to be gained by examining and comparing the communities, processes, and practices of FOSSD in different communities. This may help clarify what observations may be specific to a given community (e.g., GNU/Linux projects) compared to those that span multiple and mostly distinct communities. Two of the communities identified above are primarily oriented to develop software to support scholarly research

(scientific computing and academic software systems design) with rather small user communities. In contrast, the other two communities are oriented primarily towards software development efforts that may replace/create commercially viable systems that are used by large end-user communities. Thus, rather than focusing on a single community as the primary source for insight, reflective practice, or systematic empirical study, it is clear that FOSSD processes, practices, and projects can be studied with or across (a) disparate communities as well as (b) different application domains.

Each of these highlighted items (in italics or via Web links above) point to the public availability of data that can be collected, analyzed, and re-represented within ethnographic narratives [4, 7], computational process models [24], or for quantitative studies [16, 17]. Significant examples of each kind of data (source code and artifacts, development processes, project repositories, etc.) have been collected and analyzed by different FOSSD researchers [11]. A later section highlights different types of FOSSD studies that use such substantial, diverse, and publicly available data. The next section identifies a number of FOSSD artifact types that serve to document the social actions and technical practices that facilitate and constrain FOSSD processes.

Beyond the communities identified above, other empirical studies of FOSSD reveal that FOSSD work practices, engineering processes, and project community dynamics can best be understood through observation and examination of their socio-technical elements at different, or across multiple, levels of analysis. In particular, FOSSD projects can be examined through a *micro-level* analysis of (a) the actions, beliefs, and motivations of individual FOSSD project participants [4, 7] and (b) the social or technical resources that are mobilized and configured to support, subsidize, and sustain FOSSD work and outcomes [12, 18, 22]. Similarly, FOSSD projects can be examined through *meso-level* analysis of (c) patterns of cooperation, coordination, control, leadership, role migration, and conflict mitigation [2, 7, 15, 22] and (d) project alliances and inter-project socio-technical networking [14, 16, 23]. Last, FOSSD projects can also be examined through *macro-level* analysis of (e) multi-project FLOSS ecosystems [17] and (f) FOSSD as a social movement and emerging global culture [6]. Each of these levels of analysis can be decomposed into finer granularity topics for study. For example, micro-level studies can address topics in FOSSD including (1) motives or incentives for individual participation, (2) use of personal computing resources, (3) individual beliefs supporting free versus open source software development, (4) self-organization and self-management of competent FLOSS developers, (5) the discretionary time and effort committed by FLOSS developers, (6) trust and social accountability mechanisms, or (7) types and uses for different types of online artifacts that support

decentralized development work of project participants. We take this last topic for further elaboration in the next section. However, it should be clear that studies of FOSSD at any level of analysis will require appropriately selected research methods and data samples [11].

6.4. Informalisms for describing FLOSS requirements and design rationale

Functional and non-functional requirements for FLOSS systems are elicited, analyzed, specified, validated, and managed through a variety of Web-based artifacts. These descriptive documents can be treated as software informalisms. *Software informalisms* [22] are the information resources and artifacts that participants use to describe, proscribe, or prescribe what's happening in a FOSSD project. They are informal narrative resources codified in lean descriptions that are comparatively easy to use and publicly accessible to those who want to join the project or just browse around. An early study demonstrated how software informalisms can take the place of formalisms like “requirement specifications” or software design notations, which are documentary artifacts seen as necessary to develop high quality software according to the SE community [22]. Yet these software informalisms often capture the detailed rationale, contextualized discourse, and inquisitive debates for why changes were made in particular development activities, artifacts, or source code files. The example in Exhibit 1 displays a software informalism in the form of a threaded message discussion that describes, contextualizes, and rationalizes why a new software system version and feature set were implemented and now required.

The content such an informalism embodies requires peer review and comprehension by other developers on a project before acceptable or trusted contributions can be made or retrospectively justified [7, 15]. Finally, the choice to designate these descriptions as informalisms is to draw a distinction between how the requirements of FLOSS systems are described in contrast to the recommended use of formal requirements notations and deliverable documents (“formalisms”) that are advocated in traditional approaches [26]. However, to be clear, though SE often stresses the need to formalize software requirements specifications and designs, formalisms need not be limited to those based on a mathematical logic or state transition semantics, but can include descriptive schemes that are formed from structured or semi-structured narratives, such as those employed in Software Requirements Specifications documents. But as described below, FOSSD informalisms are generally quite different from formal SE documents.

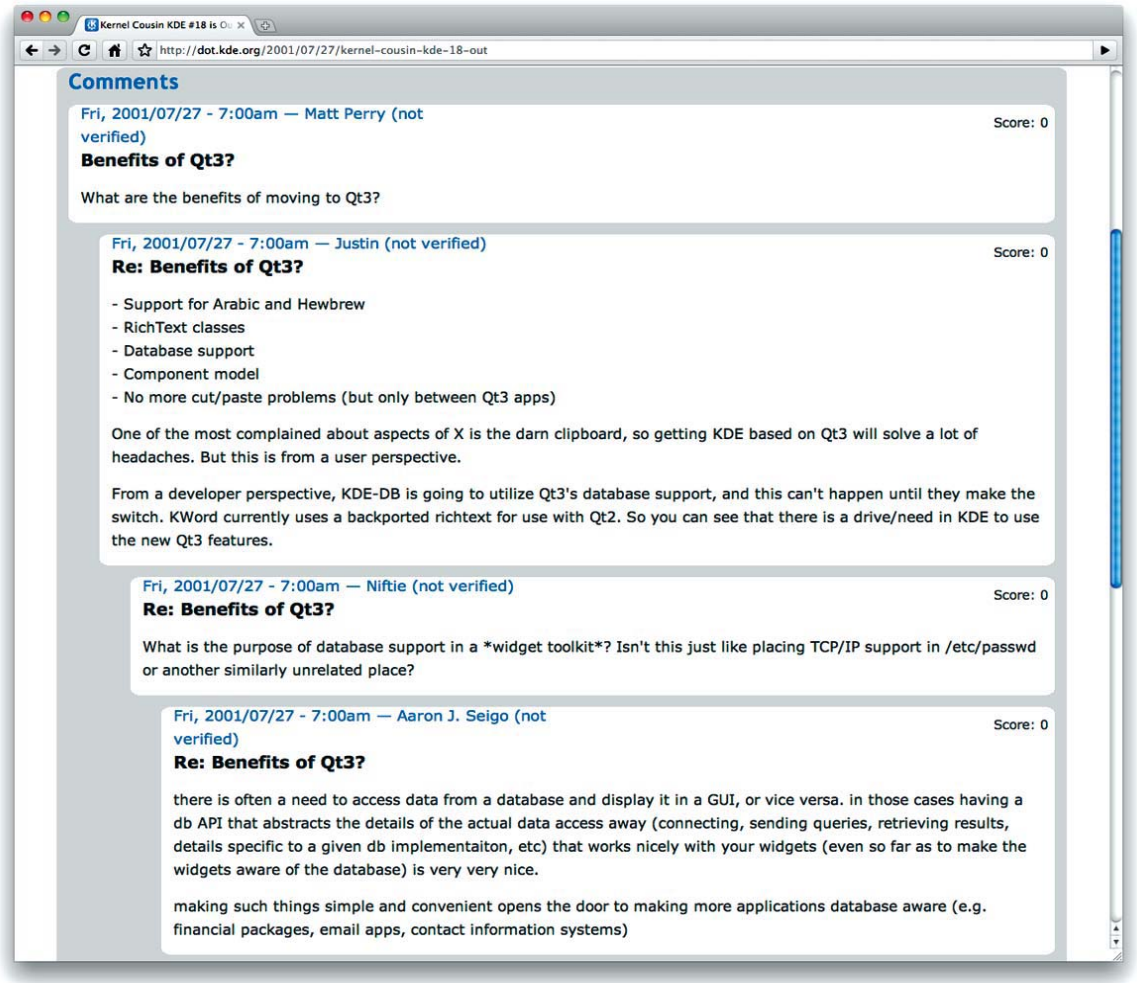


Exhibit 1. A screenshot display of a software informalism use to described, contextualize, and rationalize why a particular software feature was implemented and is thus required (Source: http://dot.kde.org/2001/07/27/kernelcousin-kde-18-out)

Two dozen types of software informalisms can be identified, and each has sub-types that can be further identified and studied more closely. Each FOSSD project usually employs only a subset as its informal document ecosystem that meets the interests or needs of local project participants. There are no guidelines or best practices for which informalisms are to be used. Instead, we have observed practices that recur across FOSSD projects. Thus, it is pre-mature and perhaps inappropriate to seek to further organize these informalisms into a classification or taxonomic scheme whose purpose is to prescribe when or where best to use one or another. Subsequently, they are presented here as an unordered list because to do otherwise would transform this descriptive information into an interpretive

scheme that suggests untested, hypothetical prescriptions [22].

The most common informalisms used in FOSSD projects include (i) communications and messages within project email, (ii) threaded message discussion forums (see Exhibit 1), bulletin boards, or group blogs, (iii) news postings, and (iv) instant messaging or Internet relay chat. These enable developers and users to converse with one another in a lightweight, semi-structured manner, and now, use of these tools is global across applications domains and cultures. As such, the discourse captured in these tools is a frequent source of FLOSS requirements. A handful of FOSSD projects have found that summarizing these communications into (v) project digests [7] helps provide an overview of major development activities, problems, goals, or debates. A project digest, such as that displayed in Exhibit 2, represents a multi-participant summary that records and hyperlinks the original contextualized rationale accounting for focal project activities, development problems, current software quality status, and desired software functionality. Project digests (which sometimes are identified as “kernel cousins”) record the discussion, debate, consideration of alternatives, code patches, and initial operational/test results drawn from discussion forums, online chat transcripts, and related online artifacts [7].

As FLOSS developers and users employ these informalisms, they have been found to also serve as carriers of technical beliefs and debates over desirable software features, social values (e.g., reciprocity, freedom of choice, freedom of expression), project community norms, and affiliation with the global FLOSS social movement [6, 10]. Other common informalisms include (vi) scenarios of usage as linked Web pages or screenshots of software in operation, (vii) how-to guides, (viii) to-do lists, (ix) Frequently Asked Questions, and other itemized lists, and (x) project Wikis, as well as (xi) traditional system documentation and (xii) external FOSSD publications [3, 9]. FLOSS (xiii) project property licenses (whether to assert collective ownership, transfer copyrights, insure “copyleft,” or some other reciprocal agreement) are documents that also help to define what software or related project content are protected resources that can subsequently be shared, examined, modified, and redistributed.

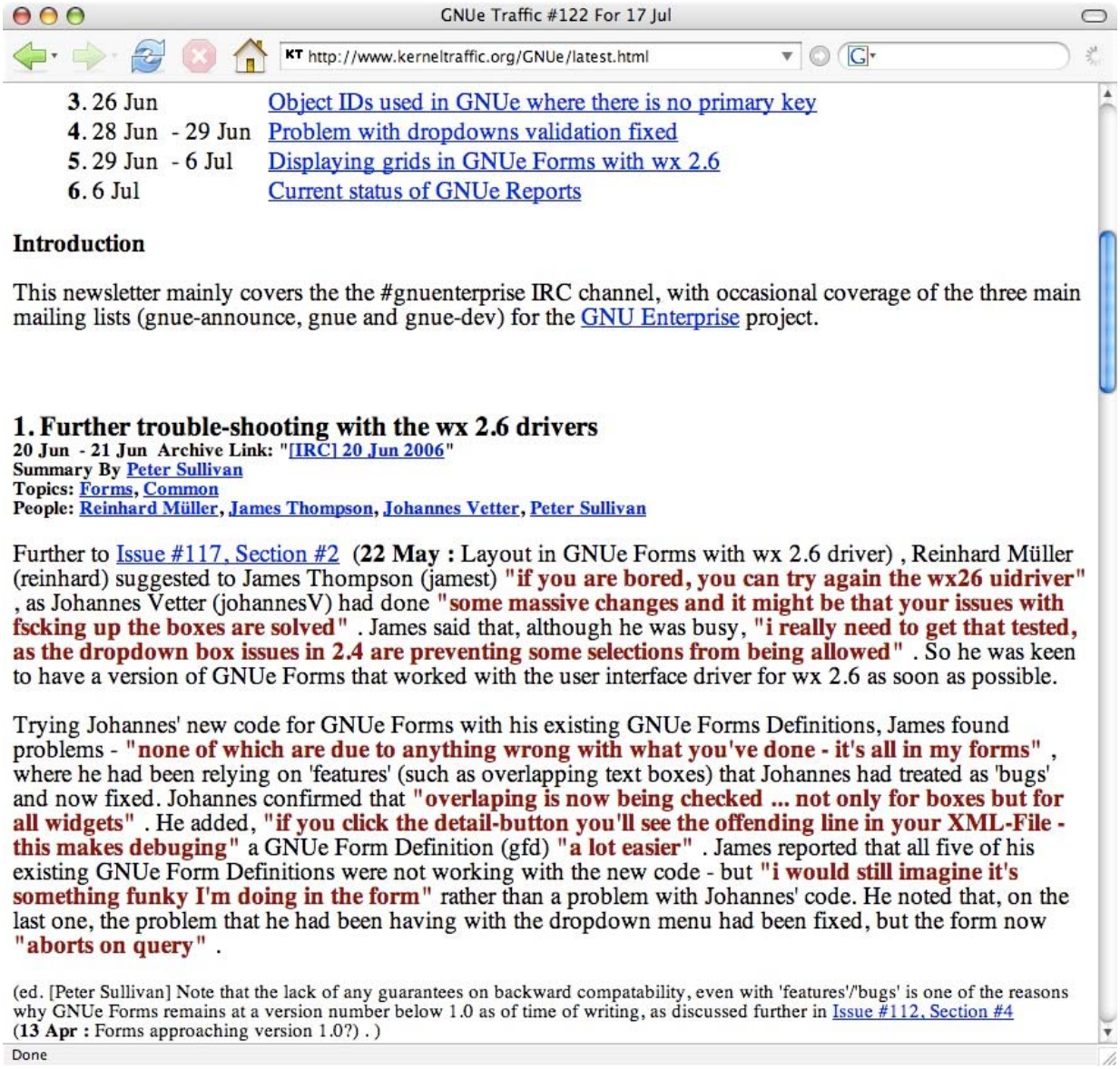


Exhibit 2. A project digest that summarizes multiple messages including those hyperlinked (indicated by highlighted and underlined text fonts) to their originating online sources. (Source: http://www.kerneltraffic.org/GNUe/latest.html, July 2006)

Next, (xiv) open software architecture diagrams, (xv) intra-application functionality realized via scripting languages like Perl or PhP, and the ability to either (xvi) incorporate externally developed software modules or “plug-ins”, or (xvii) integrate software modules from other FOSSD efforts, are all resources that are used informally where or when needed according to the interests or actions of project participants.

All of the software informalisms are found or accessed from (xix) project related Web sites or portals. These Web environments are where most FLOSS software informalisms can be found, accessed, studied, modified, cross-referenced, and redistributed.

A Web presence helps make visible the project's information infrastructure and the array of information resources that populate it. These include FOSSD multi-project Web sites (e.g., SourgeForge.net, Savanah.org, Freshment.org, Tigris.org, Apache.org, Mozilla.org, or Google.com/code), community software Web sites (PhP-Nuke.org), and project-specific Web sites (e.g., www.GNUenterprise.org) as well as (xx) embedded project source code Webs (directories), (xxi) project repositories such as those implemented using CVS or Subversion [9], and (xxii) software bug reports and (xxiii) issue tracking data bases like Bugzilla [see http://www.bugzilla.org/]. Last, giving the growing global interest in online social networking, it is not surprising to find increased attention to documenting various kinds of social gatherings and meetings using (xxiv) social media Web sites (e.g, YouTube, Google Video, Flickr, MySpace, etc.) where FLOSS developers, users, and interested others come together to discuss, debate, or work on FLOSS projects and to use these online media to record and publish photographs/videos that establish group identity and affiliation with different FLOSS projects. For example, a video presentation from 2007, http://video.google.com/videoplay?docid=-4216011961522818645 (accessed September 2008), describes how to identify and remove troublesome or unwelcome members of a FOSSD project team, who may cause the project to fragment, lose focus, become conflict laden, or otherwise fail.

In FOSSD projects, software informalisms are the preferred scheme for describing or representing FLOSS requirements, rationalizing software design choices, discussing alternative software implementations and releases, and more. There is no explicit objective or effort to treat these informalisms as "informal software requirements" that should be refined into formal requirements or design notations within these communities (the academic software design community perhaps being the exception). Accordingly, each of the available types of FOSSD informalisms have been found in one or more of the four communities identified above. Subsequently, knowledge about who is doing what, where, when, why, and how is captured in different or multiple informalisms.

Together, these two dozen types of software informalisms constitute a substantial yet continually

evolving web of informal, semi-structured, or processable information resources that capture, organize, and distribute knowledge that embody the requirements for an FOSSD project. This web results from the hyperlinking and cross-referencing that interrelate the contents of different informalisms together. Subsequently, these FLOSS informalisms are produced, used, consumed, or reused within and across FOSSD projects. They also serve to act as both a distributed virtual repository of FLOSS project assets as well as the continually adapted distributed knowledge base through which project participants evolve what they know about the software systems they develop and use.

Overall, it appears that none of these software informalisms would defy an effort to formalize them in some mathematical logic or analytically rigorous notation. Nonetheless, in the three of the four software communities examined in this study there is no perceived requirement for such formalization (academic software design being the exception) as the basis to improve the quality, usability, or cost-effectiveness of the FLOSS systems. If formalization of these documentary software informalisms has demonstrable benefit to members of the other three communities, beyond what they already realize from current practices, these benefits have yet to be articulated in the different types of software informalisms for online discourse that pervades each community. However, in contrast, the academic software design community often does encourage and embrace the development of formal requirements specification and design documents in order to coordinate and guide development of their software projects. Thus, as before, understanding FOSSD practices, processes, or projects can vary by community, such that studying such projects or associated artifacts in only one community can skew or bias what can be observed or put into practice when comparing across multiple communities. This in turn sets the stage for examining how studies of FOSSD may be organized or structured to facilitate comparative investigations.

6.5. FOSSD Research Studies and Approaches

A growing group of software researchers is beginning to investigate large software projects empirically, using freely available data from FLOSS projects. A body of recent work pointed out the need for community-wide data collections and research infrastructure to expand the depth and breadth of empirical FLOSS research, and several initial proposals have been made. Most importantly, these data collections [16, 17] and shared research infrastructure [14] are intended to support an active and growing community of FLOSS scholars addressing contemporary issues and theoretical foundations in

disciplines that include anthropology, economics, informatics (computer-supported cooperative work), information systems, library and information science, management of technology and innovation, law, organization science, policy science, and software engineering [11]. Approximately 700 published studies can be found at the FLOSShub collection of research papers, at http://flosshub.org/biblio. Furthermore, this research community has researchers based in Asia, Europe, North America, South America, and the South Pacific, thus denoting its international and global membership. Consequently, the research community engaged in empirical studies of FLOSS can be recognized as another member in the growing movement for interdisciplinary software engineering research.

This section seeks to clarify the need for community-wide, sharable research infrastructure and collections of FLOSS data. The kinds of research questions that such data sampling and corresponding research methods enable are discussed elsewhere [11].

6.5.1. Objects of study, success indicators, and critical factors for understanding FOSSD

As an organizing framework, there are primarily five main *objects of study*—that is, things whose characteristics researchers are trying to describe and explain—in FOSS-based empirical software research: *software artifacts and source code*, *software processes*, *development projects, communities*, and *participants' knowledge*. Table 1 provides a simplified set of associations between representative characteristics of FOSSD that have been investigated for each of these objects of study, and show some critical factors that researchers have begun linking to these characteristics as explanations. It is important to point out that these objects of study are by no means independent from one another. They should be considered as interdependent elements of FLOSS (e.g., knowledge and processes affect artifacts, communities affect processes, etc.). Also, each of the indicators of success shown in Table 1 may play a role as a critical factor in other categories.

6.5.2. Current FOSSD research approaches

Based on published studies of FOSSD since 2000, there are four alternative approaches in empirical research on the objects of study, measures of variables that contribute to FOSSD success, and causal mechanisms or factors that drive FOSSD success, as shown in Table 1 [11]:

Objects	Success Measures	Critical Driving Factors
Source Code and Artifacts	Quality, downloads, reliability, usability, durability, fit, structure, growth, diversity, localization	Size, complexity, bugs and features, software architecture (structure, substrates, modularity, versions, infrastructure), information architecture, artifact types, and document genres
Processes	Efficiency, ease of improvement, adaptability, effectiveness, complexity, manageability, predictability	Size, distribution, collaboration, knowledge/information management, artifact structure, configuration, agility, innovativeness
Projects	Type, size, duration, number of participants, number of software versions released	Development platforms, tools supporting development and project coordination, software imported from elsewhere, social networks, roles and role migration paths, leadership and core developers, socio-technical vision
Communities	Ease of creation, sustainability, trust, increased social capital, rate of participant turnover	Size, economic setting, organizational architecture, behaviors, incentive structures, institutional forms, motivation, participation, core values, common-pool resources, public goods
Knowledge	Creation, codification, use, need, management	Tools, conventions, norms, social structures, technical content, acquisition, representations, reproduction, application

Table 1: Characteristics of empirical FLOSS studies

- *Very large, population-scale studies* examining common objects selected and extracted from hundreds to tens-of-thousands of FLOSS projects from multi-project repositories or FLOSS research portals like FLOSSmole [14] or from international surveys of comparable numbers of FLOSS developers.
- *Large-scale cross-analyses of project and artifact characteristics*, such as code size and code change evolution, development group size, composition and organization, or development processes [16, 17].
- *Medium-scale comparative studies* across multiple FLOSS projects within different communities or software system application domains [15, 17, 18].
- *Smaller-scale in-depth case studies* of specific FLOSS practices and processes for concept or

hypothesis development and for exposing and investigating details of socio-technical interactions [4, 7, 12, 18].

These four alternative research strategies are separated less by fundamental differences in objectives than by technical limitations in existing tools and research methods, or by the socio-technical research constraints associated with quantitative studies of data mined from publicly accessible repositories versus the challenges of qualitative ethnographic research methods. For example, qualitative analyses are hard to implement on a large scale, whereas quantitative methods must rely on uniform, easily processed data. We believe these distinctions are becoming increasingly blurred as researchers develop and use more sophisticated analysis and modeling tools [25], leading to finer gradations in empirical data needs. Accordingly, attention can now turn to identifying opportunities for future research and practice that may span FOSSD and SE.

6.6. Opportunities for FOSSD and SE research and practice

There are a significant number of opportunities and challenges that arise when we look to identifying which software development or socio-technical interaction practices found in studies of FOSSD projects might be applied in the world of SE. Some of these opportunities follow. However, it is perhaps surprising to observe that it is unclear whether the world of FOSSD is interested in adopting the best practices found in the world of SE, and vice versa. For example, why would software developers seek out to engage in FOSSD projects and practices if they were the same or less than those found in SE? That is, most FOSSD projects are not seen as stepping stones to SE, nor feeble attempts at SE. Instead, FOSSD projects in most communities are seen as currently the most effective way to develop and sustain both large software systems as well as a community of like-minded developers and end-users. As such, let us consider the opportunities for what FOSSD might contribute to the world SE.

First, FLOSS poses the opportunity to favorably alter the costs and constraints of accessing, analyzing, and sharing software process and product data, metrics, and data collection instruments. FOSSD is thus poised to alter the calculus of empirical SE. Software process discovery, modeling, and simulation research [25] is one arena that can take advantage of such a historically new opportunity. Similarly, the ability to extract or data mine software product content (source code, development artifacts, etc.) within or across FLOSS project repositories to support its visualization, restructuring/refactoring, or redesign has become a high-yield, high impact area for SE study and experimentation. Another would be

examining the effectiveness and efficiency of traditional face-to-face-to-artifact SE approaches or processes for software inspections compared to the online peer reviews and multi-modal discourse informalisms prevalent in FOSSD efforts.

Second, based on results from studies of motivation, participation, role migration, and work practices of individual FLOSS developers [4, 15], it appears that the SE community would benefit from empirical studies that examine similar conditions in conventional or proprietary software development enterprises. Current SE textbooks and development processes seem to assume that individual developers have simple technical roles (e.g., software designer or programmer) and motivations driven by financial compensation and technical education, and therefore seek the quality assuring rigor that purportedly follows from the use of formal notations and mathematically based analytical schemes. Consequently, if FOSSD is more fun, interesting, and rewarding compared to SE, then what can be done to make SE more fun, interesting, and rewarding to SE students and new software developers? Said simply, SE may benefit from FOSSD practices that developers find are more fun, interesting, and rewarding, rather than merely following the 40-year legacy of SE practices, processes, and procedures prescribed in SE textbooks.

Third, based on results from studies of resources and capabilities employed to support FOSSD projects [7, 12, 18, 22], it appears that conventional software cost estimation or accounting techniques (e.g., “total cost of operation” or TCO) are limited to analyzing resources or capabilities that are easily quantified or monetized. This in turn suggests that many social and organizational resources/capabilities are slighted or ignored, thus producing results that miscalculate the diversity of resources and capabilities that affect the ongoing/total costs of software development projects whether FLOSS or SE based.

Fourth, based on results from studies of cooperation, coordination, and control in FOSSD projects [2, 12, 15, 23], it appears that virtual project management and socio-technical role migration/advancement can provide a slimmer and lighter weight approach to SE project management. However, it is unclear whether we will see corporate experiments in SE that choose to eschew traditional project management and administrative control regimes in favor of enabling software developers the freedom of choice and expression that may be necessary to help provide the intrinsic motivation to self-organize and self-manage their SE project work.

Fifth, based on results of studies on alliance formation, inter-project social networking, community development, and multi-project software ecosystems [14, 16, 17], it appears that SE projects currently operate at a disadvantage compared to FOSSD projects. In SE projects, it is commonly assumed that developers and end-users are distinct communities and that ongoing software evolution is governed by market imperatives, the need to extract maximum marginal gains (profit), and resource-limited software maintenance effort. SE efforts are setup to produce systems whose growth and evolution is limited rather than capable of sustaining exponential growth of co-evolving software functional capability and developer-user community [23].

Last, based on studies of FOSSD as a social movement [6], it appears that there is an opportunity and challenge for encouraging the emergence of a social movement that combines the best practices of FOSSD and SE. The emerging niche world of open source software engineering (OSSE) is the likely locus of collective action that might enable such a movement to arise. For example, as suggested in Exhibit 3, the community Web portal for *Tigris.org* is focused on cultivating and nurturing the emerging OSSE community—the community that spans both FOSSD and SE. Nearly 1,000 OSSE projects are currently affiliated with this portal and community. It might therefore prove fruitful to closely examine different samples of OSSE projects at *Tigris.org* to see which SE tools, techniques, and concepts are being brought to bear and to what ends in different FLOSS projects, as well as which SE concepts and methods are employed to develop FLOSS-based software development tools.

6.7. Limitations and Conclusions on FOSSD

FOSSD is certainly not a panacea for developing complex software systems, nor is it simply SE done poorly. Instead, it represents an alternative community-intensive socio-technical approach to develop software systems, artifacts, and social relationships. However, it is not without its limitations and constraints. Thus, we should be able to help see these limits as manifest within the level of analysis or research for empirical FOSSD studies examined above. First, in terms of participating, joining, and contributing to FOSSD projects, an individual developer's interest, motivation, and commitment to a project and its contributors is dynamic and not indefinite. Some form of reciprocity and self-serving or intrinsic motivation seems necessary to sustain participation in a FOSSD project, whereas a perception of exploitation by others can quickly dissolve a participant's commitment to further contribute, or worse to dissuade other participants to abandon a FOSSD project that has gone astray.

Exhibit 3. A view of the Tigris.org project community and its intent to span and bring together the FOSSD and SE communities (Source: http://www.tigris.org , accessed March 2010)

Similarly, most FOSSD projects are not open to anyone who just wants to immediately become a core developer, system architect, or group leader without going through a protracted socio-technical process that requires demonstration of technical competence as well as willingness to help other project participants and facilitate project growth in ways aligned to the dominant interests of the overall project community. Developers who want to pursue alternative approaches, software architectures, or

divisions of labor may be undesirable, unwanted, and at times be pushed out of a FOSSD project (cf. Footnote 8). FOSSD is as much about developing and sustaining the project community as it is about the software and how it is being developed.

Second, in terms of cooperation, coordination, and control, FOSSD projects do not escape conflicts in technical decision-making, or in choices of who gets to work on what, or who gets to modify and update what. Because FOSSD projects generally lack traditional project managers, they must become self-reliant in their ability to mitigate and resolve outstanding conflicts and disagreements. Beliefs and values that shape system design choices as well as choices over which software tools to use and which software artifacts to produce or use are determined through negotiation rather than administrative assignment. Negotiation and conflict management then become part of the cost that FLOSS developers must bear in order for them to have their beliefs and values fulfilled. It is also part of the cost they bear in convincing and negotiating with others often through electronic communications to adopt their beliefs and values. Time, effort, and attention spent in negotiation and conflict management represent an investment in building and sustaining a negotiated socio-technical network of dependencies.

Third, forming alliances and building community through participation, artifacts, and tools points to a growing dependence on other FOSSD projects. The emergence of non-profit foundations that were established to protect the property rights of large multi-component FOSSD project creates a demand to sustain and protect such foundations. If a foundation becomes too bureaucratic, then this may drive contributors away from a project. So, these foundations need to stay lean and not become a source of occupational careers in order to survive and evolve. Similarly, as FOSSD projects give rise to new types of requirements for community building, community software, and community information sharing systems, these requirements need to be addressed and managed by FOSSD project contributors in roles above and beyond those involved in enhancing the source code of a FOSSD project. FOSSD alliances and communities depend on a rich and growing web of socio-technical relations. Thus, if such a web begins to come apart, or if the new requirements cannot be embraced and satisfied, then the FOSSD project community and its alliances will begin to come apart.

Fourth, in terms of the co-evolution of FLOSS systems and community, as already noted, individual and shared resources of people's time, effort, attention, skill, sentiment (beliefs and values), and computing resources are part of the socio-technical web of FOSS. Reinventing existing software

systems as FLOSS coincides with the emergence or reinvention of a community who seeks to make such system reinvention occur. FLOSS systems are “common pool resources” that require collective action for their development, mobilization, use, and evolution. Without the collective action of the FOSSD project community the common pool will dry up, and without the common pool the community begins to fragment, drift away, and disappear, perhaps to search for another pool elsewhere.

Last, empirical studies of FOSSD are expanding the scope of what we can observe, discover, analyze, or learn about how large software systems can be or have been developed. In addition to traditional methods used to investigate FOSSD like reflective practice, industry polls, survey research, and ethnographic studies, comparatively new techniques for mining software repositories and multi-modal modeling and analysis of the socio-technical processes and networks found in sustained FOSSD projects [7, 15, 24] show that the empirical study of FOSSD is growing and expanding. This in turn will contribute to and help advance the empirical science in fields like SE, which previously were limited by restricted access to data characterizing large, proprietary software development projects. Thus, the future of empirical studies of software engineering practices, processes, and projects will increasingly be cast as studies of free/open source software development efforts.

Overall, one of the defining characteristics of FOSSD projects is that data about development processes, work practices, and project/community dynamics is publicly available on a global basis. Data about FOSSD products, artifacts, and other resources is kept in repositories associated with a project’s Web site, though finding and extracting it may involve substantial effort. This may include the site’s content management system, computer mediated communication systems (email, persistent chat facilities, and discussion forums), software versioning or configuration management systems, and networked file systems. FOSSD process data is generally either extractable or derivable from data/content in these artifact repositories, though not always easily, nor in forms readily amenable to systematic analysis without further processing, reformatting, and cleaning. First-person data may also be available to those researchers or students who participate in a project, even if just to remotely observe (“lurk”) or to electronically interview other participants about development activities, tools being used, the status of certain artifacts, and the like. The availability of such data suggest a growing share of empirical SE research will henceforth be performed in domains of FOSSD projects rather than using data from in-house or proprietary software development projects that have constraints on access and publication. FOSSD process data collection from publicly accessible artifact repositories will be found to be more cost-effective compared to studies of traditional closed-source, proprietary,

and in-house software development repositories [11]. Your chance to further study or participate in a FOSSD project starts from here.

Acknowledgments

Research for this article was supported by grants #0534771, #0749353, and #0808783 from the U.S. National Science Foundation; and from the Acquisition Research Program Grant #N00244-10-1-0038, and also the Center for the Edge Grant #N00244-10-1-0064, at the Naval Postgraduate School, Monterey, CA. No review, approval, or endorsement implied.

THIS PAGE INTENTIONALLY LEFT BLANK

6.8. References

1. Cambell-Kelly, M. and Garcia-Swartz, D.D., Pragmatism, not ideology: Historical perspectives IBM's adoption of open-source software, *Information Economics and Policy*, 2009, 21(3), 229–244.
2. Crowston, K., and Scozzi, B., Open Source Software Projects as Virtual Organizations: Competency Rallying for Software Development, *IEE Proceedings--Software*, 2002, 149(1), 3–17.
3. DiBona, C., Cooper, D., and Stone, M. (Eds.), *Open Sources 2.0*, 2005, O'Reilly Media, Sebastopol, CA. Also see, C. DiBona, S. Ockman, and M. Stone (Eds.). *Open Sources: Voices from the Open Source Revolution*, 1999. O'Reilly Media, Sebastopol, CA.
4. Ducheneaut, N. Socialization in an Open Source software community: A socio-technical analysis. *Computer Supported Cooperative Work*, 2005, 14(4), 323–368.
5. Eliot, L. and Scacchi, W., Developing a Knowledge-Based System Factory: Issues and Concepts, *IEEE Expert*, 1986, 1(4), 51–58.
6. Elliott, M., Examining the Success of Computerization Movements in the Ubiquitous Computing Era: Free and Open Source Software Movements, in M.S. Elliott and K.L. Kraemer (Eds.), *Computerization Movements and Technology Diffusion: From Mainframes to Ubiquitous Computing*, Information Today, Inc, Medford, NJ, 2008, 359–380.
7. Elliott, M., Ackerman, M.S., and Scacchi, W.,Knowledge Work Artifacts: Kernel Cousins for Free/Open Source Software Development, *Proc. ACM Conf. Support Group Work (Group07)*, Sanibel Island, FL, ACM Press, 2007, 177–186, November.
8. Feller, J., Fiztgerald, B., Hissam, S. and Lakhani, K. (eds.), *Perspectives on Free and Open Source Software*, 2005, MIT Press, Cambridge, MA.
9. Fogel, K. *Producing Open Source Software: How to Run a Successful Free Software Project*, 2005, O'Reilly Press, Sebastopol, CA.
10. Gay, J. (ed.), *Free Software Free Society: Selected Essays of Richard M. Stallman*, 2005, GNU Press, Free Software Foundation, Boston, MA.
11. Gasser, L. and Scacchi, W., Towards a Global Research Infrastructure for Multidisciplinary Study of Free/Open Source Software Development, in IFIP Intern. Federation Info. Processing, Vol. 275; *Open Source Development, Community and Quality*; B. Russo, E. Damiani, S. Hissan, B. Lundell, and G. Succi (Eds.), Boston, Springer, 2008,143–158.
12. German, D., The GNOME Project: A case study of open source, global software development,

Software Process—Improvement and Practice, 2003, 8(4), 201–215.

13. Goldman, R. and Gabriel, R.P., *Innovation Happens Elsewhere: Open Source as Business Strategy*, Morgan Kaufmann Publishers, San Francisco, CA, 2005.
14. Howison, J., Conklin, M., and Crowston, K., FLOSSmole: A Collaborative Repository for FLOSS Research Data and Analyses. *Intern. J. Info. Tech. And Web Engineering*, 2006, 1(3), 17–26.
15. Jensen, C. and Scacchi, W., Role Migration and Advancement Processes in OSSD Projects; A Comparative Case Study, *Proc. 29th. Intern. Conf. Software Engineering*, Minneapolis, MN, ACM Press, 2007, 364–374.
16. Lopez-Fernandez, L., Robles, G., Gonzalez-Barahona, J.M., and Herraiz, I., Applying Social Network Analysis to Community-Drive Libre Software Projects, *Intern. J. Info. Tech. and Web Engineering*, 2006, 1(3), 27–28.
17. Madey, G., Freeh, V., and Tynan, R., Modeling the F/OSS Community: A Quantitative Investigation, in S. Koch (ed.), *Free/Open Source Software Development*, IGI Publishing, Hershey, PA, 2005, 203–221.
18. Mockus, A., Fielding, R., & Herbsleb, J.D., Two Case Studies of Open Source Software Development: Apache and Mozilla, *ACM Transactions on Software Engineering and Methodology*, 2002, 11(3), 309–346.
19. Moore, J.T.S., *Revolution OS*, 2001, See http://www.revolution-os.com/ Also available for viewing at *Google Videos* and the *Internet Archive.*
20. Press, W.H., Flannery, B.P., Teukolsky, S.A., and Vetterling, W.T., *Numerical Recipes: The Art of Scientific Computation*, 1986, Cambridge University Press, New York. Also see, W.T. Vetterling, W.T., Teukolsky, S.A., Press, W.H. and Flannery, B.P., *Numerical Recipes Example Book (FORTRAN)*, 1985, Cambridge University Press, New York.
21. Scacchi, W., The Software Infrastructure for a Distributed System Factory, *Software Engineering Journal*, IEE and British Computer Society, 1991, 6(5), 355–369.
22. Scacchi, W., Understanding the Requirements for Developing Open Source Software Systems, *IEE Proceedings--Software*, 2002, 149(1), 24–39, February.
23. Scacchi, W., Free/Open Source Software Development Practices in the Computer Game Community, *IEEE Software*, 2004, 21(1), 59–67, January/February.
24. Scacchi, W., Feller, J., Fitzgerald, B., Hissam, S. and Lakhani, K, Understanding Free/Open Source Software Development Processes, *Software Process--Improvement and Practice*, 2006, 11(2), 95–105, March/April.

25. Scacchi, W., Jensen, C., Noll, J., and Elliott, M. Multi-Modal Modeling, Analysis, and Validation of Open Source Software Development Processes, *Intern. J. Internet Technology and Web Engineering,* 1(3), 49–63, 2006.
26. Sommerville, I., *Software Engineering*, 8th Edition, 2006, Addison-Wesley, New York.
27. Spencer, D.D., *Game Playing With Computers*, 1968, Spartan Books, New York.

28.

29.

30.

31.

32.

33.

34.

35.

36.

37.

38.

39. THIS PAGE INTENTIONALLY LEFT BLANK

7. Final Report Conclusions

Walt Scacchi
Institute for Software Research. University of California, Irvine USA

Thomas A. Alspaugh
Computer Science Dept., Georgetown University, Washington, DC, USA

7.1. Introduction

This chapter summarizes the overall set of results and conclusions that are documented in this Final Report for our research funded by the Acquisition Research Program at the Naval Postgraduate School through grant #N00244-10-1-0038. As noted in the beginning of this report, our research effort has culminated in five external research publications that build on results that we have originally and previously presented at the Acquisition Research Symposium in 2009 and 2010. This includes two papers published in top-tier research journals (Chapters 2 and 3), one top international research conference (Chapter 4), and two invited book chapters (Chapters 5 and 6). The detailed research results appearing in each paper in their respective chapters, follow from our original research proposal that was highlighted in Chapter 1. The remainder of this chapter further summarizes some of the main research results and identifies how they may influence policy for acquisition of software-intensive systems, or systems of systems, based on Open Architecture (OA) that can incorporate Open Source Software (OSS) components that are subject to different intellectual property (IP) licenses.

Our work presented in Chapters 2 and 4 address the fundamental questions of how software IP licenses interact in an OA OSS system to either provide the rights specified for their acquisition, in exchange for obligations acceptable for acquisition, or to result in rights and obligations that fail to meet the acquisition criteria, or in the worst case to conflict and provide no rights at all. We originated the concept of a *virtual license*, necessary to specify the rights and obligations for an OA OSS system of components not under a single license and with it give a basis for acquisition officers to specify the intellectual property rights and obligations required for a software system without tying that system to any particular existing license. This is essential in order to acquire and evolve best-of-breed systems

that can continue to be adapted to changing conditions without being constrained to a single standard software license.

In addition, as part of that work we produced and demonstrated a proof-of-concept software architecture tool with which designers and developers can calculate the virtual license that will apply to any specific software design. If that design results in a license conflict or in a virtual license that doesn't meet the system's intellectual property requirements, the tool offers automated guidance for resolving the problems. This is not a production-quality tool, but instead represents a state-of-the-art research result and proof-of-concept prototype associated with a modern doctoral dissertation in the field of Software Engineering. This tool serves its research study purposes as well as demonstrates the potential efficacy of analyzing IP license obligations and rights that can be associated with complex software-intensive systems, or systems of systems, that are composed into an OA or software product line. Furthermore, the tool establishes the theoretical groundwork on which any production-quality tool would need to be built, and demonstrates that the theory can produce results consistent with what expert human analysts would calculate, but of course the tool does so consistently in an automated and virtually instantaneous manner. Accordingly, we welcome the opportunity to continue the development and extension of this tool, or its successors, in order to demonstrate how they can be applied to analysis of other closely related problems to OA systems, including the modeling and analysis of “software security licenses” which we suggested in our research presentation at the 2010 Acquisition Research Symposium.

Our work presented in Chapter 3 takes OA OSS system research and builds on the preceding results but in a bold new direction. There, we examine how the software architecture of an OA OSS system necessarily affects, as well as reflects, the software ecosystem in which the system is produced. This new insight begins to show how the supply chains that are essential to producing and evolving a system can be steered and evolved by the acquisition requirements for the system, including its intellectual property requirements and how its OA and OSS properties are specified, as well as the actions of the designers and developers who evolve the system. These supply chains account for how the software development efforts of originating software components or system producers are composed and integrated through independent software vendors or government contractors to address the needs of system consumers. This opens the possibility for informing the acquisition process with the awareness of how specific acquisition decisions can affect the future evolution of a

system's supply chains, whether for better (from the DoD's point of view) or worse.

Finally, in Chapters 5 and 6, we presented two book chapters that describe and elaborate what constitutes OSS, not so much in terms of the technologies involved, but more in terms of the processes and informational artifacts that are created to describe, explain, or justify why OSS components or applications are developed in such non-traditional ways as compared to proprietary, closed source software systems that have dominated the overall software ecosystem for decades. Because OA systems will increasingly embrace and integrate OSS components, future acquisition practices and policies will need to be adapted to the ways and means by which OSS components are developed as well as integrated into larger, composed systems.

7.2. Emerging research opportunities that follow from our research results

The acquisition and development of complex OA systems will continue to adapt to better exploit OSS components, informal development artifacts, and the new processes and practices which produce them. Such adaptations may be needed to: (a) take advantage of the potential cost savings, rapid time-to-deploy, and rapid software adaptation through evolutionary options that OSS increasingly represent [8,9]; (b) take advantage of new opportunities for software reuse, software product lines, and new software development business practices; and (c) to provide guidance for how to design or analyze the acquisition of software-intensive systems that are composed with components that are subject to different, heterogeneous software IP licenses, which must be honored and tracked throughout their development, deployment, and evolution across the life of the system. However, from our research perspective, some new lines of acquisition research can follow.

At least two topics merit discussion following from our approach to semantically modeling and analyzing OA systems that are subject to heterogeneous software licenses identified in the previous chapters of this report. One is how our results might shed light on software systems whose architectures articulate a *software product line*, while the other is how our approach might be extended to also address the semantic modeling and analysis of *software system security requirements.*

Organizing and developing software product lines (SPLs) relies on the development and use of explicit software architectures [4,6,8]. However, the architecture of a SPL is not necessarily an OA —

there is no requirement for it to be so. Thus, we are interested in discussing what happens when SPLs may conform to an OA, and to an OA that may be subject to heterogeneously licensed SPL components. Three considerations come to mind.

- First, if the SPL is subject to a single homogeneous software license, which may often be the case when a single vendor or government contractor has developed the SPL, then the license may act to reinforce a vendor lock-in situation with its customers. One of the motivating factors for OA is the desire to avoid such lock-in, whether or not the SPL components have open or standards-compliant APIs.
- Second, if an OA system employs a reference architecture much like we have in the design-time architecture depicted in Figure 5 in Chapter 3, which is then instantiated into a specific software product configuration as suggested in the build-time architecture shown in Figure 6 in Chapter 3, then such a reference or design-time architecture as we have presented it in this report effectively defines a SPL consisting of possible different system instantiations composed from similar components instances (e.g., different but equivalent Web browsers, word processors, email, calendaring applications, relational database management systems).
- Third, if the SPL is based on an OA that integrates software components from multiple vendors or OSS components that are subject to heterogeneous licenses, then we have a situation analogous to what we have presented in this paper. So, SPL concepts are compatible with OA systems that are composed from heterogeneously licensed components.

Next, as already noted, software licenses represent a collection of rights and obligations for what can or cannot be done with a licensed software component. Licenses thus denote non-functional requirements that apply to a software systems or system components as intellectual property (IP) during their development and deployment. But rights and obligations are not limited to concerns or constraints applicable only to software as IP. Instead, they can be written in ways that stipulate non-functional requirements of different kinds. Consider, for example, that desired or necessary software system security properties can also be expressed as rights and obligations addressing system confidentiality, integrity, accountability, system availability, and assurance [4,5].

Traditionally, developing robust specifications for non-functional software system security properties in natural language often produces specifications that are ambiguous, misleading, inconsistent across system components, and lacking sufficient details [11]. Using a semantic model to formally specify the rights and obligations required for a software system or component to be secure [4,5,11] means that it

may be possible to develop both a “security architecture” notation and model specification that associates given security rights and obligations across a software system or system of systems. Similarly, it suggests the possibility of developing computational tools or interactive architecture development environments that can be used to specify, model, and analyze a software system’s security architecture at different times in its development—design-time, build- time, and run-time.

The approach we have been developing for the past few years for modeling and analyzing software system license architectures for OA systems [1,2,10] may therefore be extendable to also being able to address OA systems with heterogeneous “software security license” rights and obligations. Furthermore, the idea of common or reusable software security licenses may be analogous to the reusable security requirements templates proposed by Firesmith [7] at the Software Engineering Institute. Consequently, such an exploration and extension of the semantic software license modeling, meta-modeling, and computational analysis tools that also support software system security can be recognized as a promising next stage of our research studies as we have proposed for study in the next stage of our acquisition research in 2011.

Overall, this final report examines the role of software ecosystems with heterogeneously licensed components in the development and evolution of OA systems that include OSS components. License rights and obligations play a key role in how and why an OA system evolves in its ecosystem. We note that license changes across versions of components are a characteristic of OA systems and software ecosystems with heterogeneously licensed components. A structure for modeling software licenses and the license architecture of a system and automated support for calculating its rights and obligations are needed in order to manage a system’s evolution in the context of its ecosystem. We have outlined an approach for achieving these and sketched how they further the goal of reusing components in developing software-intensive systems. Much more work remains to be done, but we believe that this approach turns a vexing problem into one for which workable solutions can be obtained.

THIS PAGE INTENTIONALLY LEFT BLANK

Acknowledgments

This research described in this Final Report is supported by grant #N00244-10-1-0038 from the Acquisition Research Program at the Naval Postgraduate School.

THIS PAGE INTENTIONALLY LEFT BLANK

7.3. References

1. T. A. Alspaugh, H. U. Asuncion, and W. Scacchi. Intellectual property rights requirements for heterogeneously-licensed systems. In *Proc. 17th IEEE International Requirements Engineering Conference (RE'09)*, 24–33, Aug. 31–Sept. 4 2009.
2. T. A. Alspaugh, H. U. Asuncion, and W. Scacchi. Software licenses, open source components, and open architectures. In *Proc. 6th Acquisition Research Symposium,* May 2009.
3. J. Bosch. *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach*. Addison-Wesley Professional, New York, 2000.
4. T. D. Breaux and A. I. Anton. Analyzing goal semantics for rights, permissions, and obligations. In *Proc. 13th IEEE International Conference on Requirements Engineering (RE'05)*, 177–188, 2005.
5. T. D. Breaux and A. I. Anton. Analyzing regulatory rules for privacy and security requirements. *IEEE Transactions on Software Engineering*, 34(1), 5–20, 2008.
6. P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley Professional, New York, 2001.
7. D. Firesmith. Specifying reusable security requirements. *Journal of Object Technology*, 3(1), 61–75, Jan-Feb. 2004.
8. N. Guertin and P. Clements, Comparing Acquisition Strategies: Open Architecture vs. Product Lines, In *Proc. 7th Annual Acquisition Research Symposium*, May 2010.
9. S. Hissam, C.B. Weinstock, and L. Bass, On Open and Collaborative Software Development in the DoD, In *Proc. 7th Annual Acquisition Research Symposium*, May 2010.
10. W. Scacchi and T. A. Alspaugh. Emerging issues in the acquisition of open source software within the U.S. Department of Defense. In *Proc. 5th Annual Acquisition Research Symposium*, May 2008.
11. S. S. Yau and Z. Chen. A framework for specifying and managing security requirements in collaborative systems. In *Proc. Third International Conference on Autonomic and Trusted Computing (ATC 2006)*, 500–510, 2006.